Matthew Cummings
October 31, 2021
CISS 247 Computer Systems
Lab 5 Report

Introduction
In this lab we expanded on the ARM emulator we wrote in C last week to include stack operations and function support. Additionally, because I failed to include LDUR and STUR in last week's submission, both instructions were added this week to complete the required set of instructions.

Process
The same basic structure for the emulator was used, for better or for worse. Everything is mashed into main the same as before, with one massive while(fgets()) loop to iterate over the lines of assembly instruction in a file. Helper functions to process each instruction, as well as functions to return values found in given register and memory location would be an improvement. However, this would require a significant rewrite and time investment, and what I have works.

First, and explanation of my implementation of the stack. The stack pointer starts at SP = 4000, with a stack size of 100 deep. In the actual memory array, the index is that stack pointer divided by 8. This accomplishes the goal of having the stack pointer iterate over doublewords at a time, while also eliminating wasted array positions. If the stack pointer weren't modified before using it to access the memory at the stack location, mem[4000] and mem[3992] could be filled, for example, but mem[3999] through mem[3993] would be left empty and wasted. This is like how I access the memory array, with PC values of 100 and 200 accessing memory addresses of 25 and 50, respectively. In short, SP / 8 = stack index, while PC / 4 = mem address.

```
#define DELIM_LINE_END "\t\r\n\v\f" // delimiteers to parse lines of instructions
#define DELIM " #X,\t\r\n\v\f[]"    // used to get int values out of register and constant notation
#define MEM_SIZE 2000               // num of memory addresses
#define MEM_LENGTH 30               // max length of 1 mem address
#define SP 28                       // stack-pointer
#define PC 29                       // program counter
#define LR 30                       // link register
#define XZR 31                      // zero register
#define STACK_SIZE 100              // given minimum size for the stack
#define STACK_START 4000            // starting stack pointer
```

For macro definitions, I have included every important value that could be changed. Additionally, defining SP, PC, and LR like this removes the ability for me to forget which register represents which important value.

```
// boolean to keep track of if the stack pointer is being modified in order to check
// if the stack pointer was iterated outside the stack -- stack overflow and exit
bool isSPmodified = false;

// boolean to keep track if instruction is a branch and therefore the program counter
// should not be iterated and instead should just branch to given value
bool isBranch = false;

// handle SP and LR destinations
if (strcmp(dest, "SP") == 0)
{
    destI = SP;
    isSPmodified = true;
}

else if (strcmp(dest, "LR") == 0)
{
    destI = LR;
}

// only SP and LR registers are referred by alias in code, everything else
// can be handled normally by accessing the register index
else
{
    destI = atoi(dest);
}
```

Supporting the SP and LR registers being referred to by their alias names in assembly instead of X28 and X30 requires some special handling that can be seen here. Boolean isSPmodified tracks if the stack pointer has been modified per instruction, allowing a later check to see if it has been iterated out of the stack causing a stack overflow. The next bool isBranch checks if the current instruction is a branch, stopping the emulator from automatically iterating the program counter when the instruction is directly branching to a specific program counter value. Finally, setting the destination register value is handled with SP and LR by checking for each in the code, and setting the destination location accordingly.

```
// process instruction
// ARM ADDI
if (strcmp(inst, "ADDI") == 0)
{
    // get first operand
    char oper[4];
    strcpy(oper,strtok(NULL, DELIM));
    int operI = 0;

    // set operand to value in register
    // check if operand is SP
    if (strcmp(oper, "SP") == 0)
    {
        // alias for 13 which is ARM stack pointer
        operI = reg[SP];
    }

    else if (strcmp(oper, "LR") == 0)
    {
       operI = reg[LR];
    }

    else
    {
        operI = reg[atoi(oper)];
    }

    // get constant
    char con[4];
    strcpy(con, strtok(NULL, DELIM));
    int conI = atoi(con);

    // put solution in destination register
    reg[destI] = operI + conI;

}
```

Here we have the updated ADDI implementation, able to handle modifying the stack pointer and link register effectively. The only changes here were adding the same code as above to get the values of the stack pointer and link register so they may be added with the contents of the destination register. This block of code checking for if the string “oper” matches “SP” or “LR” repeats itself in further instruction implementations, such as ADD, SUBI, LDUR, etc. These instructions all need the same special handling of SP and LR because SP and LR cannot otherwise be converted to the register indexes 28 and 30 that they represent.

Were I to rewrite this emulator, I would choose to add helper functions to get values out of register and memory locations that would eliminate this repeating code problem. Instead, the int operI and destI would be set by a helper function, say getRegValue(), which would fetch the numerical value out of a given register location. This function would have contained this same code to fetch the value and would significantly improve legibility.

Additionally, each instruction could be implemented in its own separate function, such that this if-else tree would contain just a function call in each conditional statement, instead of all this code handing the full instruction in the if() statement block.

```
else if (strcmp(inst, "ADD\0") == 0)
{
    // get first operand
    char oper[4];
    strcpy(oper,strtok(NULL, DELIM));
    int operI = 0;

    // set operand to value in register
    // check if operand is SP
    if (strcmp(oper, "SP") == 0)
    {
        operI = reg[SP]; // alias for ARM stack pointer
    }

    else if (strcmp(oper, "LR") == 0)
    {
       operI = reg[LR];
    }

    // get integer value for register as long as its not SP
    else
    {
        operI = reg[atoi(oper)];
    }

    // get second operand
    char oper2[4];
    strcpy(oper2,strtok(NULL, DELIM));
    int oper2I = 0;

    // set operand to value in register
    // check if operand is SP
    if (strcmp(oper2, "SP") == 0)
    {
        oper2I = reg[SP]; // alias for ARM stack pointer
    }

    else if (strcmp(oper2, "LR") == 0)
    {
       oper2I = reg[LR];
    }

    // get integer value for register as long as its not SP
    else
    {
        oper2I = reg[atoi(oper)];
    }

    // put solution in destination register
    reg[destI] = operI + oper2I;
```

The implementation for ADD is very similar to ADDI, except the constant operand from ADDI is replaced by a second register operand to be added to the first operand. Specifically, oper2I replaces conI to result in operI + oper2I = result. The value of the result is put into the given register index location, represented by destI.

Stack pointers and link registers and handled in the same way as ADDI, expectedly, and can also be replaced by a function call given a rewrite to achieve this.

```
// ARM SUBI
else if (strcmp(inst, "SUBI") == 0)
{
    // get first operand
    char oper[4];
    strcpy(oper,strtok(NULL, DELIM));
    int operI = 0;

    // set operand to value in register
    // check if operand is SP
    if (strcmp(oper, "SP") == 0)
    {
        operI = reg[SP]; // alias for ARM stack pointer
    }

    else if (strcmp(oper, "LR") == 0)
    {
       operI = reg[LR];
    }

    // get integer value for register as long as its not SP
    else
    {
        operI = reg[atoi(oper)];
    }

    // get constant
    char con[4];
    strcpy(con, strtok(NULL, DELIM));
    int conI = atoi(con);

    // put solution in destination register
    reg[destI] = operI - conI;

}
```

The SUBI implementation is almost identical to ADDI, save for the final statement that subtracts conI from operI instead of adding.

```
// branch to link register
else if (inst[0] == 'B' && inst[1] == 'L')
{
    // branch PC to value in LR + destI value (destI represents constant)
    reg[PC] = reg[LR] + destI;

    // if branching to LR the stack needs to be cleared
    for (int j = reg[SP]; j <= STACK_START; j = j + 8)
    {
        strcpy(mem[j/8], "");
    }

    // reset stack pointer to the top
    reg[SP] = STACK_START;

    isBranch = true;
}

// branch to register
else if (inst[0] == 'B' && inst[1] == 'R')
{
    // make sure register being branched to isn't XZR
    if (dest[0] == 'Z' && dest[1] == 'R')
    {
        reg[PC] = 0;
    }

    // only use given register if register is not XZR
    else
    {
        reg[PC] = reg[destI];
    }
    isBranch = true;
}

// ARM branch
else if (inst[0] == 'B')
{
    // branch amount same as destI above
    reg[PC] = reg[PC] + destI;
    isBranch = true;
}
```

Next are the implementations of non-conditional branch. There are 3 variations of branch included: B, BR, and BL. B is simple branch, which directly branches to a PC value given by a constant in the instruction, i.e., “B 260” changes the program counter directly to 260.

BR branches to the PC contained in each register index, as in “BR X3” where X3 contains 260 branches to the instruction at PC=260. Special handling of XZR is required here, as “BR XZR” is intended to terminate the program. Only chars ‘Z’ and ‘R’ are checked for in the given register, as the ‘X’ char is truncated out by strtok() for all destination registers to directly access the index of the register.

Lastly, BL branches to the value contained in the link register plus a constant offset. For example, “BL #8”, where the link register holds 260, will branch to PC=268. Additionally, BL must flush the stack and return the stack pointer to the top, as BL is used to implement functions in assembly, and exiting a function must return the stack to an empty state.

```
// conditional branch if zero
else if (strcmp(inst, "CBZ\0") == 0)
{
    // get distance to branch from instruction
    char branch[5];
    strcpy(branch, strtok(NULL, DELIM));
    int branchI = atoi(branch);

    // destI from above already holds value that needs to be compared
    // if reg[destI] is zero, add branchI to pc
    if (reg[destI] == 0) reg[PC]+= branchI;

    isBranch = true;
}
```

Conditional branch is very similar to branch with 2 main differences. First, this will only branch if the value contained in the destI register equals 0 (reg[destI]==0). Further, CBZ uses PC relative branching, which means the constant branch amount in branchI either adds or subtracts from the PC to branch to a new instruction.

```
// load mem data into reg
else if (strcmp(inst, "LDUR") == 0)
{
    // get memory location
    char oper[4];
    strcpy(oper,strtok(NULL, DELIM));
    int operI = 0;

    // set operand to value in register
    // check if operand is SP
    if (strcmp(oper, "SP") == 0)
    {
        // alias for ARM stack pointer, contents of SP point to mem
        // location to load data from
        operI = reg[SP]/8;
    }

    // get integer value for register as long as its not SP
    else
    {
        operI = reg[atoi(oper)]/4;
    }

    // get constant mem offset
    char con[4];
    strcpy(con, strtok(NULL, DELIM));
    int conI = 0;

    // divide by doubleword if accessing SP
    if (operI == reg[SP]/8)
    {
        conI = atoi(con)/8;
    }

    // else acessing normal memory only need to divide by word length
    else
    {
        conI = atoi(con)/4;
    }

    // get full contents of memory at this location
    char memcopy[MEM_LENGTH];
    strcpy(memcopy, mem[operI + conI]);

    // skip over token containing mem address
    strtok(memcopy, DELIM);

    // load memory data into register
    reg[destI] = atoi(strtok(NULL, DELIM));

}
```

Now here is LDUR, one of 2 new instructions since last weeks ARM emulator implementation (for me). LDUR loads data into the destination register from a given memory address, accessed using indirect addressing.

With indirect addressing, a starting base address is stored as a value in a CPU register, and a constant shift amount is given to access memory locations on either side of the base address.

Here, destI is the destination register being loaded into, operI is the register index holding the value for the base address, and conI holds the constant shift amount performed on the base register to access the desired memory location.

Special handling is required here for accessing memory in the stack, as the stack pointer jumps by 8 (doubleword) indices instead of the 4 (word) jumps required for standard memory access. This is what the /4 and /8 operations are doing to conI and operI within each conditional.

```
// store reg data into mem
else if (strcmp(inst, "STUR") == 0)
{
    // get memory destination
    char oper[4];
    strcpy(oper,strtok(NULL, DELIM));
    int operI = 0;

    // set operand to value in register
    // check if operand is SP
    if (strcmp(oper, "SP") == 0)
    {
        // alias for ARM stack pointer, contents of SP point to mem
        // location to load data from
        operI = reg[SP]/8;
    }

    // get integer value for register as long as its not SP
    else
    {
        operI = reg[atoi(oper)]/4;
    }

    // get constant mem offset
    char con[4];
    strcpy(con, strtok(NULL, DELIM));
    int conI = atoi(con);

    // create string to be stored in memory
    char data[MEM_LENGTH];
    if (strcmp(oper, "SP") != 0) sprintf(data, "%d %d 0 0", operI*4, reg[destI]);
    else sprintf(data, "%d %d 0 0", operI*8, reg[destI]);

    // copy formatted string to location in mem
    strcpy(mem[operI], data);
}
```

The ARM STUR instruction is like LDUR, but simpler in implementation. STUR finds the value in reg[destI] and stores it in a memory address accessed in the same way as LDUR; indirectly. The same awkward handling for the stack pointer vs program counter is required, as the stack iterates over doublewords, and normal memory here iterates over a word.

Additionally, filling memory locations with sprint() require different statements and a conditional to get the right pointer values from operI with *4 and *8. Sprintf() is used because it was the easiest and cleanest way to fill memory locations with the same formatting as in all other memory locations, with “104 224 0 0” containing only the address and data in 2 tokens.

```
printf("%s\n\nRESULT:\n", instructionCopy);
for(int i = 0; i < 28; i++)
{
    if (reg[i] != 0) printf("X%d = %d\n", i, reg[i]);
}
printf("\n");

// print prog counter, stack pointer, and link register
printf("PC = %d\n", reg[PC]);
printf("SP = %d\n", reg[SP]);
printf("LR = %d\n", reg[LR]);

// print data stored in modifiable memory
printf("\nDATA MEMORY:\n");
i = 25;
while (strcmp(mem[i], "") != 0)
{
    printf("%s\n", mem[i]);
    i++;
}
```

Now all instruction implementations have been shown and explained, here are the statements used to show the user what is happening in the virtual ARM CPU and memory.

The first for() loop iterates over the registers and prints any values that aren’t 0 (meaning they have been modified). Then, the program counter, stack pointer, and link register are printed out.

To print out all memory data that is relevant and not all 25 locations between address 100 and 200, each memory string must be compared to and empty string. Only if the memory address is not empty will it be printed.

```
// print contents of stack, if any
printf("\nSTACK:\n");
i = STACK_START / 8;
if (reg[SP] == STACK_START && mem[STACK_START/8][0] == '\0')
{
    printf("EMPTY\n");
}

else
{
    while (strcmp(mem[i], "") != 0)
    {
        printf("%s\n", mem[i]);
        i = i - 1;
    }
}
printf("\n");
```

Now the stack is printed, providing it is not empty. If the stack is found to be empty, this emulator will say the stack is empty. If not empty, the while loop will loop over the stack until it finds an empty address location, then terminate the loop.

```
// check if the stack pointer has been modified, and if so, check if
// the SP has been iterated out of the stack, causing a STACK OVERFLOW
if (isSPmodified)
{
    // check if SP is outside specified size
    if ( (reg[SP] > STACK_START) || (reg[SP] < (STACK_START - STACK_SIZE)) )
    {
        printf("%d %d\n", STACK_START, reg[SP]);
        printf("Stack overflow detected, terminating...\n");
        return 0;
    }
}
```

Here is where the isSPmodified bool is used to only check for a stack overflow if the stack has been modified in the first place. If the stack has been modified, then this code checks if the stack pointer is outside the stack. To achieve this, the current stack pointer is compared to the starting stack pointer value, as well as the starting stack pointer value minus the total size of the stack. This ensures the stack overflow check is dynamic with multiple starting stack and stack size values defined as macros at the top of this program.

```
// if PC is ever zero (like BR XZR) halt execution
if (reg[PC] == 0)
{
    printf("PC set to 0, terminating...\n");
    break;
}

// only increment PC if PC is not zero and instruction was not a branch
else if (!isBranch)
{
    reg[PC] = reg[PC] + 4;
}
```

This is where the program counter is checked for being equal to zero, as in “BR XZR”. If “BR XZR” has been executed, the program counter will have been set to 0. If the program counter has been set to 0, then the emulator will break out of the loop and show the final memory and register values. After the PC has been checked for zero, and only if a branch instruction has not been executed, the program counter will automatically increment by 4. This stops the PC being incremented on top of a branch instruction, which is unexpected behavior.

```
// wait for user to hit zero to move to next instruction
printf("Press enter for next instruction...\n");
while (getchar() != '\n')
{
    ;
}
```

This is how I handle waiting for the user to hit enter for the next instruction to run. This is an infinite loop until the enter key is pressed. Then execution moves on to the next

```
printf("End of %s reached\n", argv[1]);

// print memory array, both data and instructions
printf("\nDATA MEMORY:\n");
i = 25;
while (strcmp(mem[i], "") != 0)
{
    printf("%s\n", mem[i]);
    i++;
}

printf("\nPROGRAM MEMORY:\n");
i = 50;
while (strcmp(mem[i], "") != 0)
{
    printf("%s\n", mem[i]);
    i++;
}

printf("\nSTACK:\n");
i = STACK_START / 8;
if (reg[SP] == STACK_START && mem[STACK_START/8][0] == '\0')
{
    printf("EMPTY\n");
}

else
{
    while (strcmp(mem[i], "") != 0)
    {
        printf("%s\n", mem[i]);
        i = i - 1;
    }
}
printf("\n");
```

instruction.
This is all the code that is run once a given code.txt file of instructions has been finished. It is the same code as before at the bottom of the while loop, but instead this shows the final resulting memory, CPU registers, and stack after all instructions have been run.

Testing

Tests were performed with these ARM instructions, shown on the right. All required instructions are represented here, as well as branching to a link register and flushing the stack. Logical functions are tested starting at 260, with CBZ branching to 272 since the value in X10 minus itself equals 0. Once branched to 272, the link register is set to go back to 264, BL #0 is called, and execution returns to 264. Instruction 264 calls BR XZR, which terminates execution. 268 is never executed due to this logic, showing that CBZ, BL, and BR XZR all work as intended. Additionally, using BL will flush the stack as expected. STUR is used to fill the stack in 232 and 240 to give values to be cleared by running LR.

```
100 512 0 0 0
104 24 0 0 0
108 22 0 0
200 ADDI X0, XZR, #100
204 LDUR X9, [X0, #0]
208 ADDI X0, X0, #4
212 LDUR X10, [X0, #4]
216 ADD X2, X0, X9
220 ADD X9, X9, X10
224 ADDI X1, XZR, #500
228 STUR X1, [X0, #8]
232 STUR X9, [SP, #0]
236 SUBI SP, SP, #8
240 STUR X1, [SP, #0]
244 SUBI SP, SP, #8
248 LDUR X3, [SP, #16]
252 ADDI X3, X3, #50
256 SUB X4, X10, X10
260 CBZ X4, #12
264 BR XZR
268 ADDI X4, X4, #141
272 ADDI LR, LR, #264
276 BL #0
```

Results
Rather than pasting a very long screenshot of all results, here is a direct clipboard paste of the output of ./ARM2 code.txt.

DATA MEMORY:
100 512 0 0 0
104 24 0 0 0
108 22 0 0

PROGRAM MEMORY:
200 ADDI X0, XZR, #100
204 LDUR X9, [X0, #0]
208 ADDI X0, X0, #4
212 LDUR X10, [X0, #4]
216 ADD X2, X0, X9
220 ADD X9, X9, X10
224 ADDI X1, XZR, #500
228 STUR X1, [X0, #8]
232 STUR X9, [SP, #0]
236 SUBI SP, SP, #8
240 STUR X1, [SP, #0]
244 SUBI SP, SP, #8
248 LDUR X3, [SP, #16]
252 ADDI X3, X3, #50
256 SUB X4, X10, X10
260 CBZ X4, #12
264 BR XZR
268 ADDI X4, X4, #141
272 ADDI LR, LR, #264
276 BL #0

200 ADDI X0, XZR, #100

RESULT:
X0 = 100

PC = 200
SP = 4000
LR = 0

DATA MEMORY:
100 512 0 0 0
104 24 0 0 0
108 22 0 0

STACK:
EMPTY

Press enter for next instruction...
204 LDUR X9, [X0, #0]

RESULT:
X0 = 100
X9 = 512

PC = 204
SP = 4000
LR = 0

DATA MEMORY:
100 512 0 0 0
104 24 0 0 0
108 22 0 0

STACK:
EMPTY

Press enter for next instruction...
208 ADDI X0, X0, #4

RESULT:
X0 = 104
X9 = 512

PC = 208
SP = 4000
LR = 0

DATA MEMORY:
100 512 0 0 0
104 24 0 0 0
108 22 0 0

STACK:
EMPTY

Press enter for next instruction...
212 LDUR X10, [X0, #4]

RESULT:
X0 = 104
X9 = 512
X10 = 22

PC = 212
SP = 4000
LR = 0

DATA MEMORY:
100 512 0 0 0
104 24 0 0 0
108 22 0 0

STACK:
EMPTY

Press enter for next instruction...
216 ADD X2, X0, X9

RESULT:
X0 = 104
X2 = 208
X9 = 512
X10 = 22

PC = 216
SP = 4000
LR = 0

DATA MEMORY:
100 512 0 0 0
104 24 0 0 0
108 22 0 0

STACK:
EMPTY

Press enter for next instruction...
220 ADD X9, X9, X10

RESULT:
X0 = 104
X2 = 208
X9 = 1024
X10 = 22

PC = 220
SP = 4000
LR = 0

DATA MEMORY:
100 512 0 0 0
104 24 0 0 0
108 22 0 0

STACK:

EMPTY

Press enter for next instruction...
224 ADDI X1, XZR, #500

RESULT:
X0 = 104
X1 = 604
X2 = 208
X9 = 1024
X10 = 22

PC = 224
SP = 4000
LR = 0

DATA MEMORY:
100 512 0 0 0
104 24 0 0 0
108 22 0 0

STACK:
EMPTY

Press enter for next instruction...
228 STUR X1, [X0, #8]

RESULT:
X0 = 104
X1 = 604
X2 = 208
X9 = 1024
X10 = 22

PC = 228
SP = 4000
LR = 0

DATA MEMORY:
100 512 0 0 0
104 604 0 0
108 22 0 0

STACK:
EMPTY

Press enter for next instruction...
232 STUR X9, [SP, #0]

RESULT:
X0 = 104
X1 = 604
X2 = 208
X9 = 1024
X10 = 22

PC = 232
SP = 4000
LR = 0

DATA MEMORY:
100 512 0 0 0
104 604 0 0
108 22 0 0

STACK:
4000 1024 0 0

Press enter for next instruction...
236 SUBI SP, SP, #8

RESULT:
X0 = 104
X1 = 604
X2 = 208
X9 = 1024
X10 = 22

PC = 236
SP = 3992
LR = 0

DATA MEMORY:
100 512 0 0 0
104 604 0 0
108 22 0 0

STACK:
4000 1024 0 0

Press enter for next instruction...
240 STUR X1, [SP, #0]

RESULT:
X0 = 104
X1 = 604
X2 = 208
X9 = 1024

X10 = 22

PC = 240
SP = 3992
LR = 0

DATA MEMORY:
100 512 0 0 0
104 604 0 0
108 22 0 0

STACK:
4000 1024 0 0
3992 604 0 0

Press enter for next instruction...
244 SUBI SP, SP, #8

RESULT:
X0 = 104
X1 = 604
X2 = 208
X9 = 1024
X10 = 22

PC = 244
SP = 3984
LR = 0

DATA MEMORY:
100 512 0 0 0
104 604 0 0
108 22 0 0

STACK:
4000 1024 0 0
3992 604 0 0

Press enter for next instruction...
248 LDUR X3, [SP, #16]

RESULT:
X0 = 104
X1 = 604
X2 = 208
X3 = 1024
X9 = 1024
X10 = 22

PC = 248
SP = 3984
LR = 0

DATA MEMORY:
100 512 0 0 0
104 604 0 0
108 22 0 0

STACK:
4000 1024 0 0
3992 604 0 0

Press enter for next instruction...
252 ADDI X3, X3, #50

RESULT:
X0 = 104
X1 = 604
X2 = 208
X3 = 1074
X9 = 1024
X10 = 22

PC = 252
SP = 3984
LR = 0

DATA MEMORY:
100 512 0 0 0
104 604 0 0
108 22 0 0

STACK:
4000 1024 0 0
3992 604 0 0

Press enter for next instruction...
256 SUB X4, X10, X10

RESULT:
X0 = 104
X1 = 604
X2 = 208
X3 = 1074
X9 = 1024
X10 = 22

PC = 256

SP = 3984
LR = 0

DATA MEMORY:
100 512 0 0 0
104 604 0 0
108 22 0 0

STACK:
4000 1024 0 0
3992 604 0 0

Press enter for next instruction...
260 CBZ X4, #12

RESULT:
X0 = 104
X1 = 604
X2 = 208
X3 = 1074
X9 = 1024
X10 = 22

PC = 272
SP = 3984
LR = 0

DATA MEMORY:
100 512 0 0 0
104 604 0 0
108 22 0 0

STACK:
4000 1024 0 0
3992 604 0 0

Press enter for next instruction...
272 ADDI LR, LR, #264

RESULT:
X0 = 104
X1 = 604
X2 = 208
X3 = 1074
X9 = 1024
X10 = 22

PC = 272
SP = 3984

LR = 264

DATA MEMORY:
100 512 0 0 0
104 604 0 0
108 22 0 0

STACK:
4000 1024 0 0
3992 604 0 0

Press enter for next instruction...
276 BL #0

RESULT:
X0 = 104
X1 = 604
X2 = 208
X3 = 1074
X9 = 1024
X10 = 22

PC = 264
SP = 4000
LR = 264

DATA MEMORY:
100 512 0 0 0
104 604 0 0
108 22 0 0

STACK:
EMPTY

Press enter for next instruction...
264 BR XZR

RESULT:
X0 = 104
X1 = 604
X2 = 208
X3 = 1074
X9 = 1024
X10 = 22

PC = 0
SP = 4000
LR = 264

DATA MEMORY:
100 512 0 0 0
104 604 0 0
108 22 0 0

STACK:
EMPTY

PC set to 0, terminating...
End of code.txt reached

DATA MEMORY:
100 512 0 0 0
104 604 0 0
108 22 0 0

PROGRAM MEMORY:
200 ADDI X0, XZR, #100
204 LDUR X9, [X0, #0]
208 ADDI X0, X0, #4
212 LDUR X10, [X0, #4]
216 ADD X2, X0, X9
220 ADD X9, X9, X10
224 ADDI X1, XZR, #500
228 STUR X1, [X0, #8]
232 STUR X9, [SP, #0]
236 SUBI SP, SP, #8
240 STUR X1, [SP, #0]
244 SUBI SP, SP, #8
248 LDUR X3, [SP, #16]
252 ADDI X3, X3, #50
256 SUB X4, X10, X10
260 CBZ X4, #12
264 BR XZR
268 ADDI X4, X4, #141
272 ADDI LR, LR, #264
276 BL #0

STACK:
EMPTY

Conclusions

I need to gain confidence using functions in C. My emulator implementation here repeats a lot of code that could be replaced by a properly implemented helper function. After having started the spell checker I think I know better how to pass arrays and modify them within a function, but I only figured that out after finishing the first iteration of this ARM emulator. If I had thought it would be time efficient to rewrite most of these instruction implementations as functions this code would have come out cleaner, but I am happy to be able to say it works. Honestly, I was scared to change too much after getting it to work.

With that said, I did not only learn something about C. I feel much more comfortable dealing with ARM instructions and have a deeper understanding of how ARM CPUs process instructions. There is a level of simple beauty in how instructions so simple in form can be built up to create programs much more complex in nature. I thought writing this emulator would be significantly more difficult than it ended up being only because the instructions themselves were simple to understand. Much of the complexity was in handling the memory and register arrays with indexes, and how to parse the instruction strings to get values.