

ECSE 324: Computer Organization

Lab 2: Basic Assembly Language Programming

Due: 11:59 PM, Tuesday, October 21

Abstract

This lab introduces a variety of challenges in assembly language programming using the [ARMv7 DE1-SoC emulation environment](#). You will work with complex arrays and loops in Part 1. Part 2 adds more complex control flow, and recursion. You will submit your code for automated grading; you will also submit a short report documenting your approach, and analysis of the performance of your various implementations.

Summary of Deliverables

- Source code for each algorithm (excepting summation, which is not assessed)
- Report, no longer than one (1) page (10 pt font, 1" margins, *no cover page*)

Please submit your source code in a single .zip archive, using the following file naming conventions:

- Archive: StudentID_FullName_Lab2_src.zip
- Code: part1.s, part2-recursive.s, part2-iterative.s

Please submit your report as a PDF, using the following file naming convention:

- Report: StudentID_FullName_Lab2_report.pdf

Grading Summary

- 75% Software test cases
- 25% Report

Changelog

- 11-Sep-2025 Initial release.
- 12-Sep-2025 Additional hints and minor template modifications.
- 26-Sep-2025 Fixed a mismatch between C prototypes and implementations in Part 2.
Fixed a bug in the Part 2 starter assembly related to instruction alignment.

Getting Started

This first exercise is ungraded and intended to get you started with a straightforward translation of an algorithm in C into assembly using a function call.

Subroutine calling convention

It is important to carefully respect subroutine calling conventions in order to prevent call stack corruption. The convention we will use for calling a subroutine in ARM assembly is as follows.

The **caller** (the code that is calling a function) must:

- Move arguments into **R0** through **R3**. (If more than four arguments are required, the caller should PUSH the arguments onto the stack.)
- Call the subroutine at label *func* using BL *func*.

The **callee** (the code in the function that is called) must:

- Move the return value, if any, into **R0**.
- Ensure that the state of the processor is restored to what it was before the subroutine call by POPping arguments off of the stack.
- Use BX LR to return to the calling code.

Note that the state of the processor can be saved and restored by pushing any of the registers **R4** through **LR** that are used by the subroutine onto the stack at the beginning of the subroutine, and popping **R4** through **LR** off the stack at the end of the subroutine.

For an example of how to perform a function call in assembly, consider the implementation of the vector dot product. Note that this code is an expansion of the example presented in 4-isa.pdf.

```
// initialize memory
n:      .word 6                // the length of our vectors
vecA:   .word 5,3,-6,19,8,12   // initialization for vector A
vecB:   .word 2,14,-3,2,-5,36  // initialization for vector B
vecC:   .word 19,-1,-37,-26,35,4 // initialization for vector C
result: .space 8              // uninitialized space for the results

.global _start                // define entry point
```

```

_start:                                // execution begins here!
    LDR    A1, =vecA                    // put the address of A in A1
    LDR    A2, =vecB                    // put the address of B in A2
    LDR    A3, n                        // put n in A3
    BL     dotp                         // call dotp function
    LDR    V1, =result                  // put the address of result in V1
    STR    A1, [V1]                     // put the answer (0x1f4, #500) in result

    LDR    A1, =vecA                    // put the address of A in A1
    LDR    A2, =vecC                    // put the address of C in A2
    LDR    A3, n                        // put n in A3
    BL     dotp                         // call dotp function
    STR    A1, [V1, #4]                 // put the answer (0x94, #148) in result+4

stop:
    B      stop                        // infinite loop!

// calculate the dot product of two vectors
// pre-- A1: address of vector a
//       A2: address of vector b
//       A3: length of vectors
// post- A1: result
dotp:
    PUSH   {V1-V3}                     // push any Vn that we use
    MOV    V1, #0                       // V1 will accumulate the product

dotpLoop:
    LDR    V2, [A1], #4                 // get vectorA[i] and post-increment
    LDR    V3, [A2], #4                 // get vectorB[i] and post-increment
    MLA    V1, V2, V3, V1               // V1 += V2*V3
    SUBS   A3, A3, #1                   // i-- and set condition flags
    BGT    dotpLoop

    MOV    A1, V1                       // put our result in A1 to return it

    POP    {V1-V3}                     // pop any Vn that we pushed
    BX     LR                           // return

```

Questions? There's a channel for that in Teams.

Ungraded Practice Exercise: Summation

To introduce you to basic assembly language programming, you'll complete a template to implement a function that calculates the sum of the elements in an array. Consider the C program below. Note that the C code compiles and runs. The use of such a *golden model* can be helpful for comparison of intermediate values when debugging your assembly.

```
int sum(int *array, int length) {
    int answer = 0;
    for (int index=0; index<length; index++)
        answer += array[index];

    return answer;
} // sum

int main(int argc, char* argv[]) {
    int a[4] = {1, 2, 3, 4};
    int b[8] = {2, 3, 5, 7, 11, 13, 17, 19};

    int a_s = sum((int *) a, 4); // 10
    int b_s = sum((int *) b, 8); // 77

    return 0;
} // main
```

Complete the assembly language below to implement the C functionality above. Note that most of the C has been copied into the template. Where assembly is missing, comments have been made to remind ****you**** of what ****you**** need to do.

```
//      int a[4] = {1, 2, 3, 4};
matrixA: .word 1, 2, 3, 4
lengthA: .word 4

//      int b[8] = {2, 3, 5, 7, 11, 13, 17, 19};
matrixB: .word 2, 3, 5, 7, 11, 13, 17, 19
lengthB: .word 8

// we'll save our results here
results: .space 8

.global _start
```

```
// Summation
// Sum the integers in the given array
// pre-- A1: address of array
// pre-- A2: length of array
// post- A1: sum of elements
sum:
    // **you** push any registers used below onto the stack
    //     int answer = 0;
    //     for (int index=0; index<length; index++)
    // **you** answer=0
    // **you** index=0
sumIter:
    // **you** index<length?
    // **you** if not, branch to sumDone
    //     answer += array[index];
    // **you** read the element at index in the array
    // **you** accumulate the answer
    // **you** index++
    B     sumIter // repeat until done!

sumDone:
    // **you** move the final answer into A1
    // **you** pop any registers pushed above
    BX    LR      // return!

_start:
//     int a_s = sum((int *) a, 4); // 10
    LDR   A1, =matrixA // put the address of A in A1
    LDR   A2, lengthA   // put the length of A in A2
    BL    sum           // call the function
    LDR   V1, =results  // put the address of Results in V1
    STR   A1, [V1]      // save the answer (in A1) in result+0

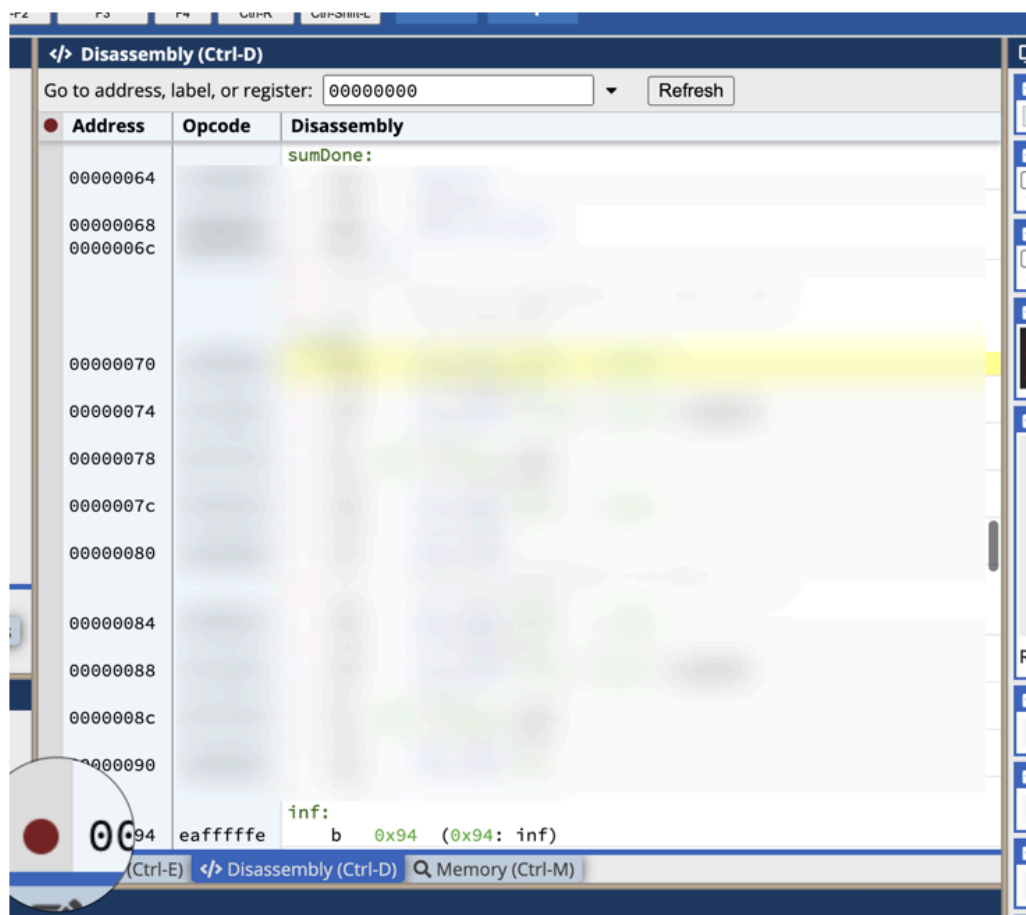
//     int b_s = sum((int *) b, 8); // 77
// **you** put the address of B in A1
// **you** put the length of B in A2
// **you** call the function
// **you** save the answer (in A1) in result+4

inf:
    B     inf          // infinite loop!
```

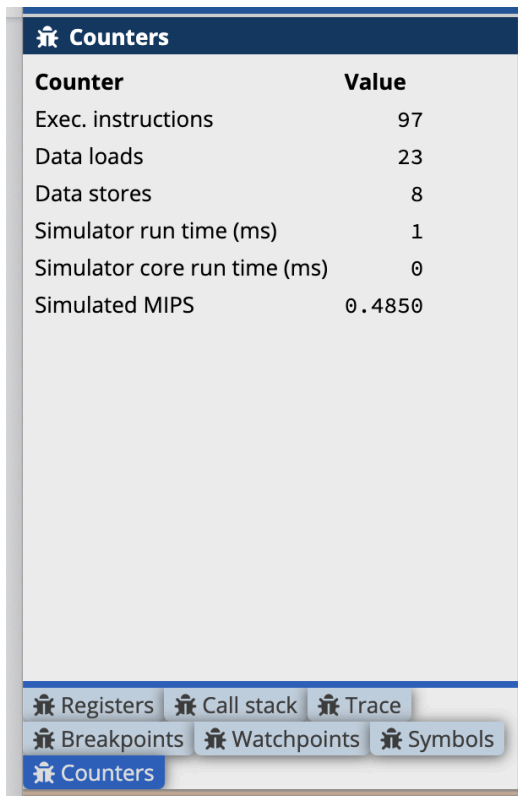
Note! Your implementation of *sum* must match the function prototype in C: use **A1** to pass the address of the array; **A2** to pass the *length* of the array; **A1** to return the *sum*.

Once you have your code running, *and getting the correct result*, let's evaluate it: a) how many instructions are executed running the assembled program? b) how many memory accesses are performed? c) how much memory is used by the assembled program?

First, set a breakpoint to stop execution at the infinite loop. You set breakpoints in the Disassembly view, clicking on the gray column to the left of the address where you want to pause. Once you've done this, if you click Continue, your program should run until it gets to the breakpoint, and then pause.



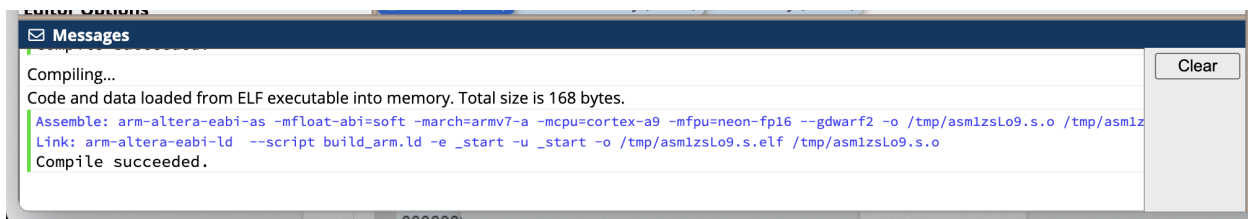
Then, to find out how many instructions were executed, amongst other things, click on the Counters pane on the left.



Counter	Value
Exec. instructions	97
Data loads	23
Data stores	8
Simulator run time (ms)	1
Simulator core run time (ms)	0
Simulated MIPS	0.4850

The first line reports the number of instructions executed so far. The next two lines report the number of data loads and stores; remember that data loads are in addition to instruction fetches. In this example, memory is accessed $97+23+8=128$ times.

Finally, to see how big the assembled program is, look in the Messages pane.



```
Messages
Compiling...
Code and data loaded from ELF executable into memory. Total size is 168 bytes.
Assemble: arm-altera-eabi-as -mfloat-abi=soft -march=armv7-a -mcpu=cortex-a9 -mfpu=neon-fp16 --gdwarf2 -o /tmp/asm1zsLo9.s.o /tmp/asm1z
Link: arm-altera-eabi-ld --script build_arm.ld -e _start -u _start -o /tmp/asm1zsLo9.s.elf /tmp/asm1zsLo9.s.o
Compile succeeded.
```

In this example, the total size of the executable (all instructions and compile-time-allocated data) is 168 bytes.

How does your program compare? Does it execute fewer instructions? Does it access memory fewer times? Does it take up less space in memory? We'll explore these metrics in more detail in the following exercises.

Part 1: Zigging and Zagging

In this part, you will translate into assembly a C implementation of zigzag array traversal. Zigzag array traversal has many uses, and is key to JPEG image compression. Assume that *main* in the C code below begins at *_start* in your assembly program. All programs should terminate with an infinite loop, like the examples above.

```
#include <stdio.h>
#include <stdlib.h>

short matrix[4][4] = { {0, 1, 2, 3}, {4, 5, 6, 7},
                       {8, 9, 10, 11}, {12, 13, 14, 15} };
short vector[4*4] = {0};

int main(int argc, char* argv[]) {
    int n = 4; // matrix dimension

    int row=0; // starting at the top left of the input array
    int col=0;
    int seq=0; // starting at the first element of the output vector
    int dir=1; // 1 for up/right, -1 for down/left

    for (; seq<n*n; seq++) {
        // copy the current element
        vector[seq] = matrix[row][col];

        // update row and col
        // moving up and right
        if (dir == 1) {
            if (col == n-1) {
                // we're at the right edge, down and turn
                row++;
                dir = -1;
            } else if (row == 0) {
                // we're along the top, right and turn
                col++;
                dir = -1;
            } else {
                // we're in the middle, continue
                col++;
                row--;
            }
        }
    }
}
```



```
        // moving down and left
    } else {
        if (row == n-1) {
            // we're at the bottom, right and turn
            col++;
            dir = 1;
        } else if (col == 0) {
            // we're at the left edge, down and turn
            row++;
            dir = 1;
        } else {
            // we're in the middle, continue
            col--;
            row++;
        }
    } // if-else
} // for

return 0;
} // int main
```

Note that the C above does not use function calls; you may do so at your discretion, but subroutines are not required for Part 1. Also note that the C code above compiles and operates correctly; it can therefore serve as a reference model for your assembly language implementations.

Implementation

To support automated grading, please use the assembly template provided below and carefully note the requirements we have for the source code you submit.

```
N:          .word 4
matrix:     .short 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
vector:     .space 32

.global _start

_start:
    // your code goes here

stop: b     stop
```

Add no code of your own that is important above `_start`: as we will strip off the top of your submission to add our test cases for auto-grading.

Hints

This algorithm uses the short (half-word) data type. The emulator will complain if you make unaligned memory accesses (i.e., if you use an address that is not a multiple of 2); be sure to adjust your test cases accordingly.

Testing

We will automatically change `N`, `matrix`, and `vector` to test your code by manipulating the first few lines of your submitted file. Be sure to devise appropriate test cases to explore the functionality of your implementation! *We will not test your code with invalid inputs, i.e., you can assume that $N > 0$, and that sufficient space is always allocated for `vector`.*

Report

Do not document your implementation of zigzag array traversal in your report.

Summary

1. Implement an assembly program that performs zigzag array traversal (**part1.s**).

Part 2: Recamán? In this economy?!

In this part, you will implement and compare two different algorithms for computing [Recamán's Sequence](#), and report on the differences in their performance.

A straightforward way to calculate the Recamán's Sequence is with [recursion](#):

```
int recaman(unsigned int num, unsigned char *array);
int search(int tgt, unsigned char *array, unsigned int size);

// calculate the num-th Recaman number and store it in array[num]
int recaman(unsigned int num, unsigned char *array) {
    // base case
    if (num == 0) {
        array[num] = 0;
        return 0;
    }
}
```

```
    unsigned int prev = 0;
    int rnums = 0;
    unsigned int rnuma = 0;

    // calculate recaman(num-1)
    prev = recaman(num-1, array);

    // calculate the two possible next numbers
    rnums = prev - num;
    rnuma = prev + num;

    // check rnums > 0 and not already in the sequence
    if ((rnums > 0) && (search(rnums, array, num-1) < 0))
        array[num] = rnums;
    else
        array[num] = rnuma;

    return array[num];
} // recaman

int search(int tgt, unsigned char *array, unsigned int size) {
    int idx = -1;

    for (int i=0; i<size; i++) {
        if (array[i] == tgt) {
            idx = i;
            break;
        } // if
    } // for

    return idx;
} // search
```

In assembly, you will write a recursive implementation, and an iterative implementation.

How can you tell if your implementation is actually recursive? A recursive function calls itself (using branch and link instruction BL) until it reaches a base case, at which point results are aggregated as each function call returns (using instruction BX). An iterative function does not call itself, and only returns (BX) once the complete result has been calculated; their behavior is best characterized as a loop.

Hints

When a function (caller) calls a function (callee), it must push any registers that may be overwritten by the callee. In general, this means the caller saves LR. In the case of recursive functions, it is often critical that the caller also save A1-A4; the callee will assume these can be overwritten (as per the ARM Procedure Call Standard). *Navigating this is one of the principal challenges of implementing truly recursive software in assembly.*

Once you've written and tested your implementation of the search function above, it should work for both your recursive and iterative solutions. *Code reuse!*

Implementation

Use the following starter code to begin your implementations.

Add no code of your own that is important above `_start`: as we will strip off the top of your submission to add our test cases for auto-grading.

```
.global _start

N:                .word 20    // input parameter n
SEQ:              .space 21    // Recaman sequence of n+1 elements
                  .space 3     // for correct alignment of instructions

_start:
    ldr           A1, N        // get the input parameter n
    ldr           A2, =SEQ     // get the address for results
    bl           recaman      // go!

stop:
    b             stop

recaman:
    // your code starts here
```

Testing

We will automatically change N and SEQ to test your code by manipulating the first few lines of your submitted file. Be sure to devise appropriate test cases to explore the functionality of your implementation! *We will not test your code with invalid inputs, i.e., you can assume that $N \geq 0$ and that space is allocated for SEQ.*

If the emulator tells you that a register has been clobbered (unexpectedly overwritten, not restored to previous state, etc), that's an indication of an error! If we observe this when running your code, marks will be deducted accordingly.

Report

Iterative implementations tend to (dramatically) out-perform recursive implementations. Do you observe that to be the case in your ARM assembly implementations? *For full marks, justify your claims.* How much of a performance difference is there between the two implementations, in terms of a) number of executed instructions, b) number of memory accesses, and c) code size? What factors do you think are contributing to these differences? It is worth exploring how the input Recamán number index n affects the metrics above.

Summary

1. Implement and evaluate an assembly program that calculates Recamán numbers recursively (**part2-recursive.s**). Collect performance metrics (code size, instructions executed, total memory accesses).
2. Implement and evaluate an assembly program that calculates Recamán numbers iteratively (**part2-iterative.s**). Collect performance metrics.
3. Compare the above in your report in no more than one written page.

Deliverables

Your code will be auto-graded as outlined above: the first few lines of your program will be automatically stripped off, and test data prepended to the file. Your program will then be run, and the contents of memory and registers will be automatically evaluated for correctness.

Your report is limited to two pages, one each for the performance analyses related to Part 2 and Part 3. Use no smaller than 10 pt font, no narrower than 1" margins, submit no cover page. The report will be graded by assessing, for each part, your report's clarity, organization, and technical content. Report grades will be modulated based on the correct operation of the submitted software. (You cannot receive full marks on a report about broken code.)

Grading

Source code

- 30% Part 1: Zigzag array traversal
- 25% Part 2A: Recursive Recamán
- 20% Part 2B: Iterative Recamán

Note that multiple test cases will be used to evaluate each submitted program; the value of each test case is weighted equally.

Report

- 25% Part 2: Recamán Performance Analysis

Each part of the report will be graded for: (a) clarity, (b) organization, and (c) technical content:

- 1pt *clarity*: grammar, syntax, word choice
- 1pt *organization*: clear narrative flow from brief overview to results, analysis, conclusions, etc.
- 3pt *technical content*: appropriate use of terms, description of results, analysis, etc.

Submission

Please submit your source code in a single .zip archive, using the following file naming conventions:

- Archive: StudentID_FullName_Lab2_src.zip
- Code: part1.s, part2-recursive.s, part2-iterative.s

Please submit your report as a PDF, using the following file naming convention:

- Report: StudentID_FullName_Lab2_report.pdf