

# ECSE 324: Computer Organization

## Lab 1: Basic Assembly Language Programming

Demos: Week of February 6  
Report: Due Friday, February 10, at 11:59 pm

### Abstract

This lab introduces a variety of challenges in assembly language programming. You will work with arrays, loops, and basic arithmetic in Part 1. Part 2 adds more complex control flow, and recursion. You will demonstrate your code live, where you step through it, showing results in registers and memories; 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)
- Demo, no longer than five (5) minutes (**week of February 6th**)
- Report, no longer than four (4) pages (10 pt font, 1" margins, *no cover page*) (**due February 10th at 11:59 pm**)

Please submit the above in a single .zip archive, using the following file name conventions:

- Code: part1-mm.s, part1-wmm22.s, part2-iterative.s, part2-recursive.s
- Report: StudentID\_FullName\_Lab1\_report.pdf

### Grading Summary

- 50% Demo
- 50% Report

### Changelog

- 5-Jan-2023 Initial revision.
- 13-Jan-2023 Typo in summation exercise (C code) corrected.
- 17-Jan-2023 Bug in binary search C code corrected. `lowIdx == highIdx` should be `lowIdx >= highIdx`.
- 20-Jan-2023 Performance analysis section of Part 2 updated to eliminate references to algorithms from Part 1.

## Part 1: The Matrices! They're Multiplying!!

In this part, you will translate into assembly two different implementations of matrix multiplication in C, and then investigate their performance. Assume that *main* in the C code below begins at *\_start* in your assembly program. All programs should terminate with an infinite loop, as in exercise (5) of Lab 0.

### *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 vector dot product. Note that this code is an expansion of the example presented in 2-isa.pdf.

```
.global _start                                // define entry point

// 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
```

```

_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

// 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.

```
.global _start

//      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
```

```

// 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

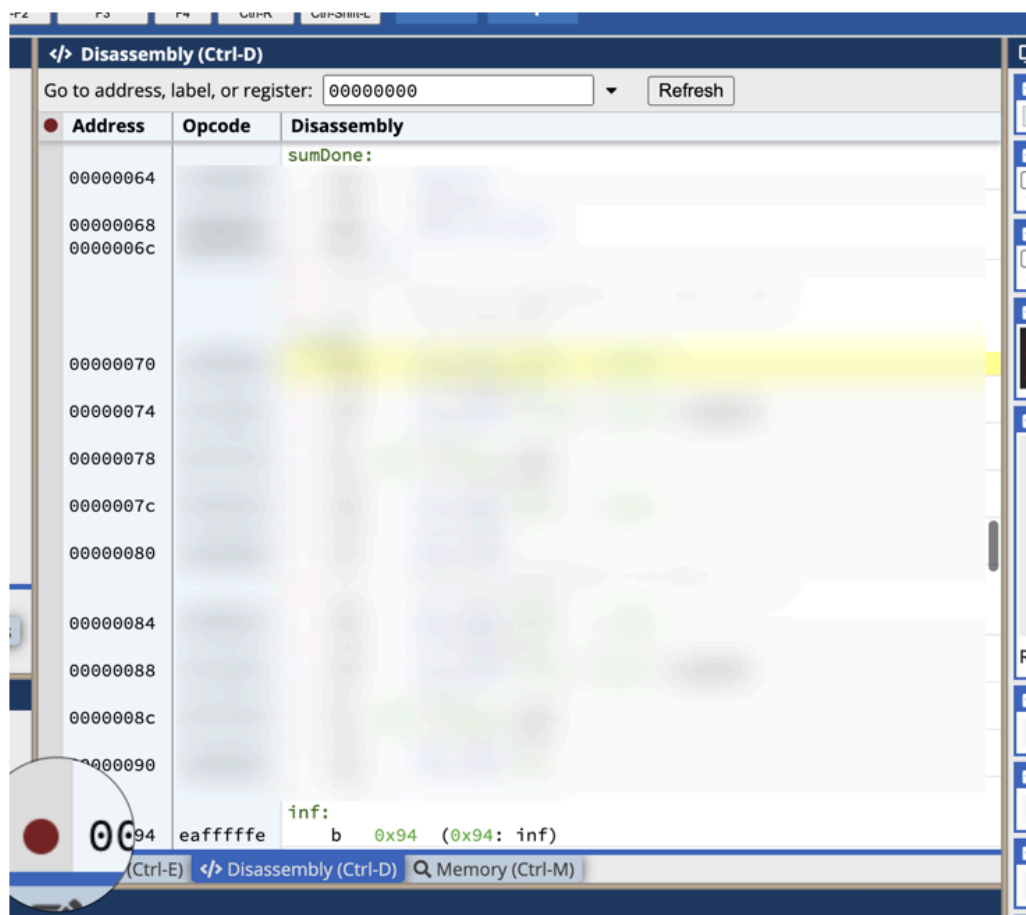
inf:
    B      inf          // infinite loop!

```

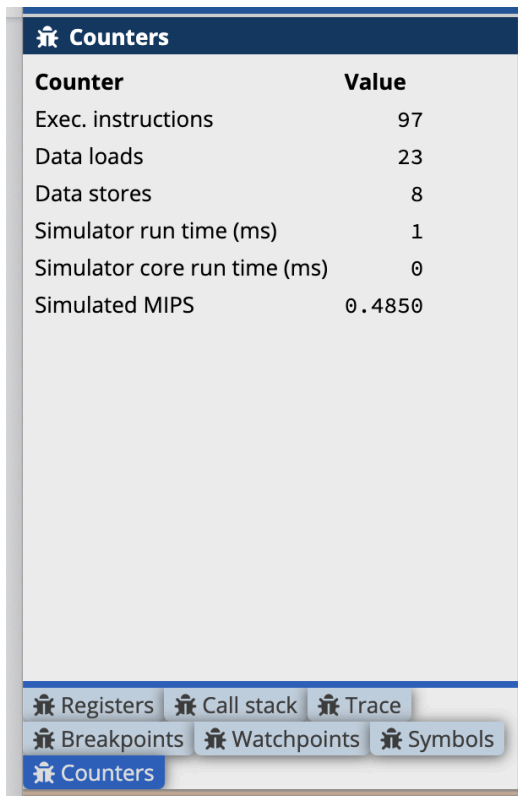
**Note!** Your implementation of *sum* must match the function prototype in C: use **A1** to pass the address of the array; **A2** to pass the *length* of the array; **A1** to return the *sum*.

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.



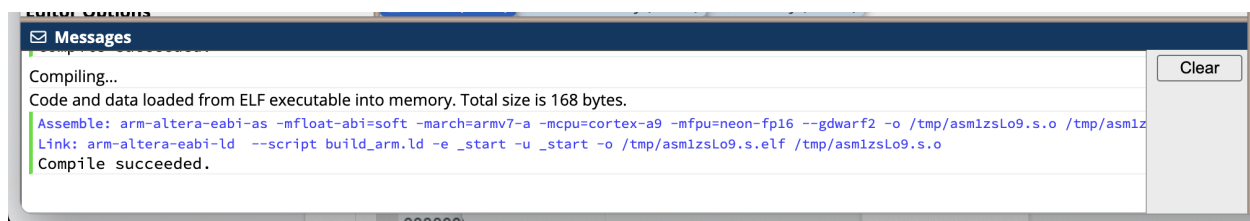
Then, to find out how many instructions were executed, amongst other things, click on the Counters pane on the left.



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.



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.

*The above exercise will not be presented during your demo, or documented in your report.*

## Matrix multiplication

Now you'll write a program from scratch that implements a straightforward matrix multiplication function, written in C below. As above, you'll need directives that define the arrays, and assembly at `_start` that sets up and performs the function call.

```
#include <stdint.h>

void mm(int16_t *a, int16_t *b, int16_t *c, unsigned int size) {
    // for each row in c
    for (unsigned int row=0; row<size; row++) {
        // for each column in c
        for (unsigned int col=0; col<size; col++) {
            *(c + row*size + col) = 0;
            // for each element in the selected row and column
            for (unsigned int iter=0; iter<size; iter++) {
                *(c + row*size + col) += \
                    *(a + row*size + iter) * \
                    *(b + iter*size + col);
            } // for
        } // for
    } // for
} // mm

int main(int argc, char* argv[]) {
    int16_t a[2][2] = {{-1, 2}, {3, -4}};
    int16_t b[2][2] = {{6, -3}, {2, 4}};
    int16_t c[2][2] = {{0, 0}, {0, 0}};
    unsigned int size = 2;

    // {{-2, 11}, {10, -25}}
    mm((int16_t *) a, (int16_t *) b, (int16_t *) c, size);

    return 0;
} // main
```

**Note!** Your implementation of `mm` must match the C function prototype above: use **A1** to pass the address of matrix `a`; **A2** to pass the address of matrix `b`; **A3** to pass the address of matrix `c`; **A4** to pass the length of one side of the square arrays, `size`.



Likewise, **note that the above implementation is using 16-bit integers, otherwise known as halfwords or shorts**. Your implementation must do the same: allocate matrices made up of shorts rather than words, and use appropriate memory access instructions.

Collect and be prepared to report a) code size, b) instructions executed, and c) total memory accesses when running your implementation; you will need them for comparison later.

### **Winograd Matrix Multiplication**

Much research has been conducted to reduce the number of arithmetic operations required in matrix multiplication. The [Winograd form](#) for 2x2 matrices, for instance, reduces the number of additions/subtractions from 18 to 15. **Write an assembly implementation of the Winograd form, in C below, and compare its performance with the standard algorithm above.**

```
void wmm22(int16_t *a, int16_t *b, int16_t *c) {
    // access each element of a and b
    // value <= *(base address + row number * row size + col number)
    int16_t aa = *(a + 0*2 + 0);
    int16_t bb = *(a + 0*2 + 1);
    int16_t cc = *(a + 1*2 + 0);
    int16_t dd = *(a + 1*2 + 1);

    int16_t AA = *(b + 0*2 + 0);
    int16_t CC = *(b + 0*2 + 1);
    int16_t BB = *(b + 1*2 + 0);
    int16_t DD = *(b + 1*2 + 1);

    // create the various intermediate values we need
    int16_t u = (cc - aa)*(CC - DD);
    int16_t v = (cc + dd)*(CC - AA);
    int16_t w = aa*AA + (cc + dd - aa)*(AA + DD - CC);

    // combine them to set the values of each element in c
    *c = aa*AA + bb*BB;
    *(c + 0*2 + 1) = w + v + (aa + bb - cc - dd)*DD;
    *(c + 1*2 + 0) = w + u + dd*(BB + CC - AA - DD);
    *(c + 1*2 + 1) = w + u + v;
} // wmm
```

**Note!** Your implementation of *wmm22* must match the function prototype above, including using appropriate data types. As before, collect and be prepared to report a) code size, b) instructions executed, and c) total memory accesses when running your implementation.

You may find you are running out of registers to use. E.g., if you use a single register for each array element in matrices  $A$  and  $B$ , you will have few left for the actual calculations! You have a few options.

1. Repeat calculations or memory accesses. E.g., you need  $cc$  for  $u$ ,  $v$ , and  $w$  (and  $cc + dd$  for  $v$  and  $w$ ), but rather than assigning it to a register and loading it once, you may choose to load it multiple times. This increases code size and memory accesses, but may reduce code complexity and ease debugging.
2. Save intermediate values on the stack. Compilers deal with running out of registers by saving values that are needed again later on the stack. You can, too! Store them, use the register for something else, and load them later.

### **Performance Analysis**

The Winograd form is intended to improve the efficiency of matrix multiplication. 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?

**Hint!** While the Winograd form requires fewer additions/subtractions, there are other important differences between the two algorithms as written above.

*For bonus credit (up to 5%) on your report, optimize one of the two implementations, and repeat the above analysis.* More points will be awarded for objectively better implementations.

**Hint!** Common approaches to optimization include:

- *Loop unrolling* (e.g., making the straightforward algorithm, with triply nested for loops, look more like the Winograd form, with all calculations listed in order, reducing the number of branch instructions, etc.)
- *Common subexpression elimination* (e.g., factoring out a calculation that is required multiple times, performing it just once, and saving the value in a register or on the stack)

### **Summary**

1. Implement and evaluate an assembly program (**part1-mm.s**) standard matrix multiplication. Collect performance metrics (code size, instructions executed, total memory accesses).
2. Implement and evaluate an assembly program (**part1-wmm22.s**) Winograd matrix multiplication. Collect performance metrics.
3. Compare the above; for a bonus, optimize one or the other, and compare again.

## Part 2: To Search, or Not To Search: It's Binary

In this part, you'll implement two versions of [binary search](#): one using an iterative approach, another using recursion. As in Part 1, write a “main” function at `_start` that sets up a sorted array to search, calls your binary search function, and then enters an infinity loop.

### *Iterative Binary Search*

Translate the C function below and implement it as part of a program that calls it.

```
int binarySearch(int *array, int x, unsigned int lowIdx, unsigned int
highIdx) {
    while (1) {
        // are we done?
        if (lowIdx >= highIdx) {
            // yep!
            if (x == array[lowIdx])
                // found it!
                return lowIdx;
            else
                // didn't find it :(
                return -1;
        } // if

        // find the index in the middle of low and high indices
        unsigned int mid = (lowIdx + highIdx)/2;

        // early stopping; did we get lucky?
        if (x == array[mid])
            // yep!
            return mid;
        else
            // no, we didn't: adjust low or high index
            if (x > array[mid])
                // x is to the right
                lowIdx = mid + 1;
            else
                // x is to the left
                highIdx = mid - 1;
    } // while
} // binarySearch
```

**Note!** Your implementation of *binarySearch* must match the function prototype above. Use an array of eight (8) elements. As in Part 1, collect and be prepared to report a) code size, b) instructions executed, and c) total memory accesses when running your implementation. You will compare these with another implementation of binary search.

**Hint!** Your ARM processor does not implement division. What instruction might you use to divide by 2?

### *Recursive Binary Search*

Now write another implementation of binary search using [recursion](#). A recursive function is a function that uses the stack to call itself until a base case is reached. The above C has been re-written below.

```
int binarySearch(int *array, int x, unsigned int lowIdx, \
                unsigned int highIdx) {
    // are we done?
    if (lowIdx >= highIdx) {
        // yep!
        if (x == array[lowIdx])
            return lowIdx; // found it!
        else
            return -1; // didn't find it :(
    } // if

    // find the index in the middle of low and high indices
    unsigned int mid = (lowIdx + highIdx)/2;

    // early stopping; did we get lucky?
    if (x == array[mid])
        return mid; // yep!
    else
        // no, we didn't; adjust low or high index
        if (x > array[mid])
            // x is to the right
            return binarySearch(array, x, mid+1, highIdx);
        else
            // x is to the left
            return binarySearch(array, x, lowIdx, mid-1);
} // binarySearch
```

**Note!** Again, ensure your implementation matches the function prototype above. The function *calls itself*, a hallmark of recursion. This means that your implementation will include BL instructions (not just B instructions, as in the iterative implementation); because BL uses the link register **LR**, you'll need to PUSH {LR} before *binarySearch* calls *binarySearch*, and POP it after the nested call returns.

As above, use an array of eight (8) elements, and collect and be prepared to report a) code size, b) instructions executed, and c) total memory accesses when running your implementation.

### ***Performance Analysis***

Recursion is a useful abstraction, but recursive implementations often perform more poorly than iterative ones. Do you observe this to be the case? *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?

**Hint!** The obvious difference between the two is that the iterative version uses a loop, while the recursive one uses function calls. What consequences does this have?

*For bonus credit (up to 5%) on your report, optimize one of the two implementations, and repeat the above analysis.* More points will be awarded for objectively better implementations.

**Hint!** Beyond the optimization approaches in Part 1, [tail call elimination](#), for instance, is commonly applied to recursive functions.

## Deliverables

Your demo is limited to 5 minutes. It is useful to highlight that your software computes correct partial and final answers; draw our attention to the registers and memory contents at appropriate points to demonstrate that your software operates as expected.

Your demo will be graded by assessing, for each algorithm, the correctness of the observed behavior, and the correctness of your description of that behavior.

In your report, for each algorithm, describe:

- Your approach (e.g., how you used subroutines, including passing arguments and returning results, how you used the stack, etc)
- Challenges you faced, if any, and your solutions
- Shortcomings, possible improvements, etc

Your report is limited to four pages, total (no smaller than 10 pt font, no narrower than 1" margins, no cover page). It will be graded by assessing, for each algorithm, your report's clarity, organization, and technical content.

## Grading

Your demo and report are equally weighted. The breakdown for the demo and report are as follows:

### *Demo*

- 25% Part 1A: Matrix multiplication
- 25% Part 1B: Winograd matrix multiplication
- 25% Part 2A: Iterative binary search
- 25% Part 2B: Recursive binary search

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

- 1pt *clarity*: the demo is clear and easy to follow
- 1pt *technical content*: correct terms are used to describe your software
- 3pt *correctness*: given an input, the correct output is clearly demonstrated

### *Report*

- 20% Part 1A: Matrix multiplication
- 20% Part 1B: Winograd matrix multiplication
- 10% Part 1 Performance analysis
- 20% Part 2A: Iterative binary search
- 20% Part 2B: Recursive binary search
- 10% Part 2 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 problem description, approach, testing, challenges, etc.
- 3pt *technical content*: appropriate use of terms, description of proposed approach, description of testing and results, etc.

## Submission

Please submit, on MyCourses, your source code, and report, in a single .zip archive, using the following file name conventions:

- Code: `part1-mm.s`, `part1-wmm22.s`, `part2-iterative.s`, `part2-recursive.s`
- Report: `StudentID_FullName_Lab1_report.pdf`