ECSE 324: Computer Organization
Lab 1: Basic Assembly Language Programming

Demos: Week of February 7
Report: Due Friday, February 11, at 11:59 pm

Abstract

This lab presents three different programming problems: (a) function calls, and recursive function calls, and (b) sorting arrays. 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.

Summary of Deliverables

- Source code for each algorithm
- Demo, no longer than five (5) minutes (week of February 7th)
- Report, no longer than four (4) pages (10 pt font, 1” margins, *no cover page*) (due February 11th at 11:59 pm)

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

- Code: `part_1-exp.s`, `part_1-fact.s`, `part_1-expbysq.s`, `part_2.s`
- Report: `StudentID_FullName_Lab1_report.pdf`

Grading Summary

50%	Demo
50%	Report

Changelog

21-Jan-2022	Initial revision.
28-Jan-2022	Due dates for various deliverables prominently added.
3-Feb-2022	Clarification added regarding function prototypes, arguments, and return values.

Part 1: Calling all functions!

In this part, your task is to implement a number of different relatively straightforward functions. 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 which 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 *func* using BL *func*.

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

- Move the return value into R0.
- Ensure that the state of the processor is restored to what it was before the subroutine call by POPing 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 R4 through LR 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.

Exponential function

You'll start with a function that calculates x^n using a loop as follows in the C code below. Note that you need not explicitly implement a function called *main*.

```c
int exp(int x, int n) {
    int result = 1;
    for (int i=0; i<n; i++)
        result = result * x;

    return result;
} // exp

int main(int argc, char* argv[]) {
    int a, b;

    a = exp(2, 10); //  1024
    b = exp(-5, 5); // -3125
} // main
```

Note! Your implementation of *exp* must match the function prototype above: use **R0** to pass *x*; use **R1** to pass *n*. Return the result in **R0**.

Factorial function

Now you will implement a function that calculates *n!* using recursion. A recursive function is a function that uses the stack to call itself until a base case is reached. The C below describes a recursive function that calculates the factorial function for a given positive integer *n*:

```c
int fact(int n) {
    if (n < 2)
        // base case
        return 1;
    else
        return n * fact(n-1);
}

int main(int argc, char* argv[]) {
    int a, b;

    a = fact(5); // 120
    b = fact(10); // 3628800
} // main
```

For example, fact(4) is computed as follows:

5 * fact(4) = 5 * 4 * fact(3) = 5 * 4 * 3 * fact(2) = 5 * 4 * 3 * 2 * fact(1) = 5 * 4 * 3 * 2 * 1 = 120.

Note! Your implementation of *fact* must match the function prototype above: use **R0** to pass *n*. Return the result in **R0**.

Exponentiation by Squaring

Now that we have seen how recursion works, we can implement exponentiation by squaring, a more efficient technique for calculating x^n.

```
int exp(int x, int n) {
    int result = 1;

    // base cases
    if (n == 0)
        return 1;
    if (n == 1)
        return x;

    // check if n is even or odd
    if (n & 1)
        // n is odd
        return x * exp(x * x, n >> 1);
    else
        // n is even
        return exp(x * x, n >> 1);
} // exp

int main(int argc, char *argv[]) {
    int a, b;

    a = exp(2, 10); //  1024
    b = exp(-5, 5); // -3125
} // main
```

Note! Your implementation of *exp* must match the function prototype above: use **R0** to pass *x*; use **R1** to pass *n*. Return the result in **R0**.

Exercises

1. Write an assembly program (**`part1-exp.s`**) that computes x^n for any integer *x* and any positive integer *n*. *exp* must be implemented as a function that is called by your program *more than once*.
2. Write an assembly program (**`part1-fact.s`**) that uses recursive function calls to calculate *n!*. *fact* must be implemented as a function that is called by your program *more than once*.
3. Write an assembly program (**`part1-expbysq.s`**) that uses recursive function calls to compute x^n for any integer *x* and any positive integer *n* using exponentiation by squaring. *exp* must be implemented as a function that is called by your program *more than once*.

Part 2: Sort this out quickly

In this part, your task is to implement the quicksort algorithm. Quicksort is a recursive algorithm; you will additionally implement a helper function that swaps elements in an array.

```
/* swap to elements in the array */
void swap(int *array, int a, int b) {
    int temp;

    temp = array[a];
    array[a] = array[b];
    array[b] = temp;
} // swap

/* main quicksort algorithm */
void quicksort(int *array, int start, int end) {
    int i, j, pivot, temp;

    if (start < end) {
        pivot = start;
        i = start;
        j = end;

        while (i < j) {
            // move i right until we find a number greater than the pivot
            while (array[i] <= array[pivot] && i < end)
                i++;
            // move j left until we find a number smaller than the pivot
            while (array[j] > array[pivot])
                j--;

            // swap the elements at these positions
            // unless they are already relatively sorted
            if (i < j)
                swap(array, i, j);
        } // while

        // swap pivot and element j
        swap(array, pivot, j);

        // recurse on the subarrays before and after element j
        quicksort(array, start, j-1);
        quicksort(array, j+1, end);
    } // if
} // quicksort

/* program initializes an array and calls quicksort */
int main(int argc, char* argv[]) {
    // initialization
    int length = 10;
    int numbers[10] = {68, -22, -31, 75, -10, -61, 39, 92, 94, -55};
    int *ptr = &numbers[0];

    quicksort(ptr, 0, length-1);
    // output: numbers = {-61, -55, -31, -22, -10, 39, 68, 75, 92, 94}
} // main
```

Note! Your implementation of *quicksort* must match the function prototype above: use **R0** to pass a pointer to an array; use **R1** to pass the lowest index to be sorted; use **R2** to pass the highest index to be sorted.

Exercise

1. Write an assembly program (**`part2.s`**) that implements the quicksort algorithm to sort the given array in ascending order. quicksort must be implemented as a function that is called by your program.

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

- 10% Part 1A: Exponential function
- 20% Part 1B: Factorial function
- 30% Part 1C: Exponentiation by squares
- 40% Part 2: Quicksort

Each section will be graded for (a) clarity, (b) technical content, and (c) correct execution:

- 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

- 10% Part 1A: Exponential function
- 20% Part 1B: Factorial function
- 30% Part 1C: Exponentiation by squares
- 40% Part 2: Quicksort

Each section 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: `part_1-exp.s`, `part_1-fact.s`, `part_1-expbysq.s`, `part_2.s`
- Report: `StudentID_FullName_Lab1_report.pdf`