CS 382 Fall 2023 — Computer Architecture and Organization — Project 1

Contents

This is an **individual** project.

This project is designed to challenge you to be better. That being said, this *is* a difficult project, so **please start early**. Even though you should do your best to finish the project, we understand that some of you might not be able to, and it's totally ok! That's why we put more points on the basic tasks.

1 Background

In C language, when we want to print a variable out, we can simply use `printf()`. However, if we want to print an array out, that wouldn't be as easy as in Python, and we will have to use a loop to print each element individually. You, an ambitious and talented CS-382 student, decided to make a revolution on C language by providing a new function that can print an array out with just one call. To play a twist on `printf()`, you call this function `pringle()`.

We declare `pringle()` function as follows:

```
1 unsigned long int pringle(char*, ...);
```

This is very similar to `printf()`, where the first parameter is always a string that contains zero or more format specifiers (such as `%d`). The rest of the parameters are denoted as `...`, meaning it can have any number and any

type of parameters. In case you're wondering, this is called *variadic* function. Implementing this in C involves new knowledge and is not the purpose of this project. However, using all we have learned so far, we can easily implement this in assembly.

1.1 Examples

Let's first see some examples of what this function can do. Assume we have the following code in C:

```
unsigned long int arr[] = {10, 20, 30};
pringle( "Print an array: %a\n", arr, 3 );
```

where `%a` is the specifier for our `pringle()` function. The output will look like this:

```
Print an array: 10 20 30
```

Another example:

```
unsigned long int arr1[] = {10, 20, 30, 40};
unsigned long int arr2[] = {100, 200, 300};
pringle("First array: %a\nSecond array: %a\n", arr1, 4, arr2, 3);
```

The output will look like this:

```
First array: 10 20 30 40
Second array: 100 200 300
```

As you see from the examples above, one specifier `%a` corresponds to two parameters: the array pointer, and the length of the array:

```
pringle(char* formatted_str,
        unsigned long int* ptr_arr1, unsigned long int len_arr1,
        unsigned long int* ptr_arr2, unsigned long int len_arr2, ...);
```

1.2 Assumptions

To make sure you understand the main points of the project without getting tangled in too much details, we make the following assumptions throughout the project:

(1) We will only work on `unsigned long int` arrays;
(2) The maximal number of `%a` in one `pringle()` call is 3 (see extra credits);
(3) The number of arrays passed to `pringle()` always matches the number of `%a`'s in the string;
(4) The maximal length of outputted string is the number used in the starter code;
(5) No need for error checking. For example, the array length is always correct.

We split the project into four smaller parts, and grade them individually to maximize your points. Each task will be the building block of a complete `pringle()` function. Please write each task with **correct calling conventions** from the beginning, because it'll save you a lot of time, headache, and tears when you put them together.

2 Description

2.1 Task 1 (30 pts): Number to ASCII

Let's start with the smallest task — convert an integer to ASCII, the opposite operation of the one you wrote in homework 2. In this task, you can assume all the numbers are unsigned so no need to consider negative numbers. The function is declared as follows:

```
char* itoascii(unsigned long int number);
```

2.2 Task 2 (30 pts): Returning a String of Numbers

The procedure of is declared as follows:

```
char* concat_array(unsigned long int* arr, unsigned long int len);
```

This procedure will concatenate all the elements in an array pointed by `arr` of length `len`, and return the pointer to this new string. For example, the following call

```
unsigned long int arr[] = {10,20,30};
char* ret = concat_array(arr, 3);
```

will return a string of the following

```
10 20 30
```

with a space between every two numbers (without newline at the end). To simplify implementation, you can certainly have an additional space after the last number.

Note that this function will use a loop and call `itoascii()` on each number. Also, it is highly recommended that you clear out the string in `.data` segment before you do anything in this procedure.

2.3 Task 3 (30 pts): Counting Specifiers

Because `pringle()` is variadic, there's uncertain number of parameters, and it'll be difficult to retrieve them from registers/stack. However, in our project, the first parameter is always a string with a certain number of format specifiers (`%a`). Thus, we can first count the numbers of specifiers in the string and to know how many parameters.

Note that one `%a` needs **two** parameters: the pointer to the array, and the length of the array.

This procedure is declared as follows:

```
unsigned long int count_specs(char* str);
```

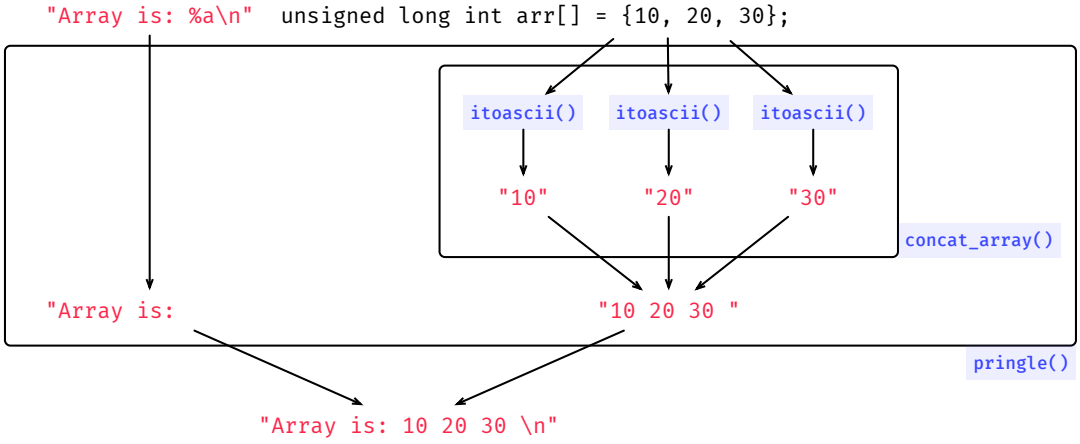
where the function will return the number of `%a`'s in `str`.

2.4 Task 4 (10 pts): Mingle and Pringle!

Last task is to complete `pringle()` function. After completing the three tasks above, you'll see the algorithm is very simple:

(1) Scan every character in the string;
  (a) If there's a `%a`:
    i. Fetch the corresponding array pointer and its length;
    ii. Call `concat_array()`;
    iii. Copy the return of `concat_array()` into the destination string;
  (b) Else copy the character from the `pringle()` parameter to the destination string;
(2) Return the number of characters printed (excluding null terminator).

We use the following flowchart to visualize the execution of `pringle()` function.



2.5 Extra Credits (20 pts)

If it's not challenging enough for you, an obvious opportunity to earn extra credits is to remove the limit of parameters we can pass to `pringle()` function. We will grade this part manually. Please indicate if you want to be graded for EC at the top of `pringle.s` as a comment.

3 Testing

Another skill you will gain from this project is to link your C code with assembly code. From a practical aspect, for some functions the compiler cannot produce the most efficient assembly code, and that's where you, a radical and intelligent CS-382 student, rise and shine! You can implement the function in assembly, and call the function from your C code directly.

3.1 Individual Tests

The first test file we provide is `indietest.c`. From this file, you can see how we declare functions written in assembly, and how to call them. First of all, we declare those functions as `extern`, since they're only defined during the linking phase:

```
1 extern char*    itoascii(unsigned long int);
2 extern char*    concat_array(unsigned long int*, unsigned long int);
3 extern unsigned long int count_specs(char*);
```

```
extern unsigned long int pringle(char*, ...);
```

We then provide a mini main function where we test all four tasks:

```
/* Test for task 1.
   Output: 1234
 */
char* s = itoascii(1234);
puts(s);

/* Test for task 2
   Output: 10 200 30
*/
unsigned long int arr[] = {10,200,30};
char* x = concat_array(arr, 3);
puts(x);

/* Test for task 3
   Output: 3
 */
char* str = "Hello this is a test string for %a and %%%a and %% and %a.\n";
unsigned long int c = count_specs(str);
char* count = itoascii(c);
puts(count);

/* Test for task 4
   Output: Hello this is a test string for 123 456 7890  and %%12  and %% and 0 0 0 0 5 .
           79
 */
unsigned long int arr1[] = {123,456,7890};
unsigned long int arr2[] = {12};
unsigned long int arr3[] = {0,0,0,0,5};
unsigned long int ret = pringle(str, arr1, 3, arr2, 1, arr3, 5);
char* newcount = itoascii(ret);
puts(newcount);
```

If you have finished task 1 and want to test it only, you can comment out the code for task 2, 3, and 4, and run the following commands to compile:

```
$ aarch64-linux-gnu-gcc indietest.c itoascii.s -g
```

and run `a.out`.

This file can be used as an example showing you how we call and test your functions in the tester, introduced below.

3.2 Tester

Ah, the (in)famous tester again! We highly recommend you use `indietest.c` first to make sure it's working before using the tester. The usage of the tester is the same as in previous two homework. Put this tester file in the same directory as your assembly code, and go ahead and run the tester:

```
$ qemu-aarch64 -L /usr/aarch64-linux-gnu/ tester -t
↪ {itoascii|concat_array|count_specs|all}
```

4 Grading

4.1 General Requirements

- Each procedure should be written in the file of the same name. For example, `itoascii()` should be written in the file called `itoascii.s`. If a helper function is used, please do not create any new file – just write it in the file where it's used;
- If you need to declare any global variable, feel free to declare it in the `.data` segment (if any), or just declare a new `.data` segment in the file it's used;
- If there's already some data declared in the starter file, please do not change it, and use the label to store your result;
- **Office Hours:** We ask you to do the following before seeking help during office hours or via email:
  - Comment every line of your code (except the lines to exit your program): do not just describe what the instruction does (we all know what it does), but how it's related to your algorithm. For example, we all know that `MOV X0,0` is to move 0 to `X0`, but a more appropriate comment should look like, "Initializing the index";
  - Use `gdb` first: if there's a segmentation fault, find out which line caused this. If you are not getting correct value, find out which instruction is not getting the correct value, and what your expected value is;
  - Fill in the table in the appendix (see last part): because the project is very large and the register use can be very messy, we ask you to keep track of your register use in the table, so that our CAs can understand your program faster and easily.

  In general you need to provide three things: your code, `gdb` result, and register table. Missing any of them will significantly delay our response. We enforce this requirement to make sure everyone's time — yours, the CAs', all other students' — are well utilized and respected. Thank you for your understanding and cooperation.

4.2 Rubrics

The project will be graded based on a total of 100 points.

- Task 1 (30 pts): 10 test cases in total, **3** points each;
- Task 2 (30 pts): 10 test cases in total, **3** points each;
- Task 3 (30 pts): 10 test cases in total, **3** points each;
- Task 4 (10 pts): 5 test cases in total, **2** points each.

After accumulating points from the testing above, we will inspect your code and apply deductibles listed below. The lowest score is 0, so no negative scores.

- Task 1 – 3 (30 pts each):
  - **-30:** the code does not compile, or executes with run-time error;
  - **-30:** the code is generated by compiler;
  - **-30:** used any external libraries and/or functions (*e.g.,* `printf()`);
  - **-30:** `concat_array()` did not call `itoascii()`;

- **-20:** not managing stack frames and/or not following calling conventions, including *self-defined helper functions* if any;
- **-5:** no pledge and/or name in assembly file;

► Task 4 (10 pts):

- **-10:** the code does not compile, or executes with run-time error;
- **-10:** the code is generated by compiler;
- **-10:** used any external libraries and/or functions (*e.g.,* `printf()`);
- **-10:** `pringle()` did not call `concat_array()`;
- **-8:** not managing stack frames and/or not following calling conventions, including *self-defined helper functions* if any;
- **-1:** no pledge and/or name in assembly file;

4.3 Early Bird Extra Credits

To encourage you to start the project early (for your own good), we have different tiers of early bird extra credits for the project. Submitting N days before the deadline will earn $N\%$ extra. Note that $N \leq 5$, meaning if you submit 6 or more days before the deadline, you will get at most 5%.

Deliverable

Only submit four files: `itoascii.s`, `concat_array.s`, `count_specs.s`, and `pringle.s`. No need to zip. **Note:** if you resubmit, please resubmit ALL files.

Appendix A Register Tracking (Very Important)

The main goal of this project is to let you understand calling conventions in assembly. As you'll write a lot of procedures and also call them, it becomes extremely difficult to debug (and write code) if you don't follow the convention from the start. In the following, we provide a worksheet to plan your register use, from task 1 to task 4. When you finish a task, make sure it works first using the `indietest.c` we provide, and immediately fill in the table below to keep track of all the registers you used/modified. Then when you move on to the next task, you'll have an idea which registers you need to save on the stack, and which register do not.

Registers	Task 1 `itoascii()`	Task 2 `concat_array()`	Task 3 `count_specs()`	Task 4 `pringle()`
X0				
X1				
X2				
X3				
X4				
X5				
X6				
X7				
X8				
X9				
X10				
X11				
X12				
X13				
X14				
X15				
X16				
X17				
X18				
X19				
X20				
X21				
X22				
X23				
X24				
X25				
X26				
X27				
X28				
X29				
X30				