f2022-public-labs

Lab 10: Functions in assembly

Late penalty: 20% off, flat, after your section's deadline, until the Gradescope late deadline. After that, your score is zero.

In this lab we will first discuss the RISCV32 calling convention or ABI (application binary interface), which allows us to write functions that properly invoke each other in machine code level. Along with them we will also discuss some aspects of stacks.

Then we will write some assembly code to call standard C library functions like `malloc` and `qsort` to practice the calling convention.

Speaking of functions, we must not forget recursion! You will also code up some recursive assembly functions to compute yet again for fibonacci series and implement a recursive binary search.

The points will be evenly distributed among the 4 questions with each problem having 30 testcases. So for each question, your score will be `PASS_COUNT/30 * 25` round to the hundredths place.

1. What is a calling convention?

Suppose your friends Alice and Bob (why Alice and Bob?) wrote two assembly functions `foo(a, b)` and `bar(c)` respectively.

Now suppose you want to write yet another assembly function `baz` that will call these functions. How would you start?

We first might want to know how we should pass the arguments into `foo(a, b)` and `bar(c)`. Do we put `a` in register `x10` or `x11`? What about argument `b`? How can we make sure all our registers' values stay the same after calling `foo` and `bar`? Also, where is the return value for each of these functions?

1.1 Calling convention to the rescue!

To make our lives (and the compiler's) easier, we would want to define a uniform way to pass function arguments, preserve register values, and get the function's return value, all of which is laid out in our **calling convention**.

You have already encountered the RISC-V calling convention in the previous two labs, where you were asked specifically to get function arguments from registers `x10` and `x11`, and we put our return value in register `x10`. You can learn more about the various type of registers RISC-V provides from your data card. A brief list of the registers is provided below:

1. Argument registers (`x10-x17` / `a0-a7`)
    i. Used for passing function arguments and function value returns.
    ii. Suppose a C function `int foo(int a, int b, int c, int d)` :
        a. `a` : reg `x10`
        b. `b` : reg `x11`
        c. `c` : reg `x12`
        d. `d` : reg `x13`
        e. The mapping keeps continue until the eighth argument in `x17`
        f. `return value` : reg `x10`
    iii. Since C does not support multi-value return, `x10` is enough for returning function results.
    iv. Note these registers are caller-saved so if you use any of them in the caller function prior to calling a subroutine, you will need to save it onto the stack (more about this later).
2. Caller-saved temporary registers (`x5-x7` / `t0-t2`)
    i. These registers, if used by caller (the function calling another function), should be saved to memory (typically stack) before calling any subroutine.
    ii. This means that if `foo` calls `bar` and `foo` uses `x6` , it will need to store `x6` to the memory right before calling `bar` .
3. Callee-saved registers (`x18-x27` / `s2-s11`)
    i. These registers, if used by callee (the function being called), should be saved to memory before any modification is made.
    ii. Suppose `foo` calls `bar` and `bar` uses `x19` , `bar` will need to save it before changing its value and restore it at the end of function.
4. Registers with special purposes (`x1/ra` , `x2/sp`)
    i. `x1` or `ra` is the link register that usually store the function return address so that when function returns, it will return to its caller.
    ii. `x2` or `sp` is the stack pointer register. It points to where the value of a specific Procedure are being help.

As you can see, a unified calling convention for our ISA allows all programs to speak the same “language” so that they can interact with each other while making safe assumptions about what registers they can use, and how to use them.

1.2 Calling functions and returning in RISCV32 ISA

1.2.1 Calling a function

Suppose we have a function `int add_one(int a)` and we would like to call it in RISC-V assembly. How should we do this? There are 5 steps to keep in mind:

1. Save any caller-saved registers in use to the memory
2. Prepare the argument registers
3. Call the function
4. Put the return value in the argument register
5. Restore the caller-saved registers
6. Read the return value

Step 2 and 5 can be optional if you do not use the caller-saved registers, which we will discuss in section 1.3.

Ignoring step 2 and 5, the assembly code to call this function looks like this:

```
// Suppose we call the above function `int add_one(int a)` with `a = 1`

// Prepare arguments. Only need `x10`, since the function has a single argument
li x10, 1
```

```
// Call the function with `jal x1, add_one`, which stands for jump and link
// The operands are the return address register, and the function label (resolves to the address of the function)
// Review the previous lab if you are not familiar with the term `label`
jal x1, add_one

// After the function returns, the program counter will
// be set to the instruction immediately after the `jal` instruction

// Here we can stored the return value of `add_one`
// to some memory address in `x7`
sw x10, 0(x7)
```

The key instruction here is the `jal x1, label`. What it does is that it will change the `PC` value to the address at `label` and save the `PC + 4` (the instruction immediately after the `bl`) value into register `x1` or `ra` link register. Thus the CPU will know where to go back when returning from a function call.

1.2.2 Entering in a function and returning from it

Now suppose we have the reversed situation here: we have an assembly function `int sub_two(int a)`. How should we prepare for when this function is called? How should we return from it?

Similar to calling a function, this has 6 steps:

1. Save link register
2. Save callee-saved registers
3. The actual function content
4. Restore the callee-saved registers
5. Restore the link register
6. Return to address in link register

If your function does not call other functions, you do not need to save and restore the link register. This is because when calling another function, the value within the link register will be overwritten. If we do not save it properly, we lose our ability to return to the caller, and we will end up with an infinite loop in the program:

```
inf_loop_func:
   // Some assembly code without saving register `ra`
   jal x1, some_func
   add x10, x10, 1
   // Some assembly code without restoring register `ra`
   ret

// This will become an infinite loop as immediate after calling
// some_func, the `ra` register will have the address
// of `add x10, x10, 1` to it. Since we did not save
// nor restore `ra`, when we execute `ret`, we will
// also return to the instruction `add x10, x10, 1`.
// Thus forming an infinite loop.
```

If your function does not modify the callee-saved registers `s2-s11`, you do not need to save and restore them.

The returning of function is accomplished by `ret`. It will return to the address in the register, which is default to `ra`.

1.3 Utilizing the stack

In this section, we will briefly discuss what the stack is and how to use it to save and restore registers' value to and from memory.

1.3.1 Stack 101

> Note: the stack we are discussing here is not the software-based data structure, but rather an implementation of it in hardware!

```
------------- ----- 0xFFFF
|   Stack   |
|-----------|
|     |     |
|     V     |
|           |
|    Free   |
|   Memory  |
|           |
|    /|\    |
|     |     |
|-----------|
|    Heap   |
|-----------|
|    Data   |
|-----------|
|    Text   |
------------- ----- 0x0000
```

The above code block shows a typical memory arrangement for a program.

- The `Text` section (`.text` label) is where the actual program lives.
- The `Data` section (`.data` label) is where we put our global data (e.g. string literals or large constants).
- The rest of the memory space is then shared by `Heap` and `Stack` .

`Heap` is basically a memory portion that grows upward (its "pointer" to the end of it will increase). Most of the time it will be used by a memory allocator to perform dynamic memory allocation (i.e. `malloc`).

`Stack` is the opposite of `Heap` that it grows downward (from higher address to lower address): each time we put something on the stack, the stack pointer `sp` should be decremented. The common practice is to use `Stack` for storing function states (callee/caller-saved registers) and local variables within functions. Therefore you will sometimes see the term **stack frame**, which refers to the stack space allocated to a function:

```
------------- ----- 0xFFFF
|   Func_1  |
|-----------|
|   Func_2  |
|-----------|
|   Func_3  |
|-----------|
|     |     |
|     V     |
|           |
|  Rest of  |
|   Memory  |
|    ...    |
```

Here is what the stack will look like if `Func_1` called `Func_2` , and then `Func_2` called `Func_3` .

1.3.2 Using the stack in RISCV32 ISA

In the RISC-V ISA, there does not exist a `pop` or `push` instruction for removing or adding items from/to the stack. Instead, you will need to manually manage the stack space for each function call. To do so, we will need to subtract the stack pointer `sp` as we enter a function call. (Look back at the diagram if you're wondering why we're decrementing - we're moving backward!) Then, we save registers onto the newly allocated space. At the end, just before we return, we will need to restore the registers based on the store sequence and add to the stack pointer so that it moves up. This is probably best demonstrated with the following assembly code snippet:

```
// We will store x11-x16 and x1 onto the stack at the beginning
// Then we will restore them at the end
// The stack memory after storing will look like this:
//    x10
//    x11
//    x12
//    x13
//    x14
//    x15
//    x16
//    x1 <-- sp after assigning space
func:
   // Assign space to store 8 32-bit register
   // Noted we need to assign at 4-byte granularity
   // (modify the `sp` reg at multiples of 4)

   // Move our stack pointer way down
   addi sp, sp, -32
   // Then, put x1 into the space currently pointed to
   sw x1, 0(sp)
   // Then, put x16 into the space currently
   // pointed to, but add 4 to the pointer
   sw x16, 4(sp)
   // Then, put x15 into the space currently
   // pointed to, but add 8 to the pointer
   sw x15, 8(sp)
   sw x14, 12(sp)
   sw x13, 16(sp)
   sw x12, 20(sp)
   sw x11, 24(sp)
   // Then, put x10 into the space currently
   // pointed to, but add 28 to the pointer
   sw x10, 28(sp)

   // Some code

   // Now we will restore the registers
   // by the sequence they are saved
   // Be aware that stack is LIFO (last in first out)

   // Our stack pointer is at the same spot,
   // so we can use the same offsets.
   // eg. we load x1 from where the stack pointer is pointing.
   lw x1, 0(sp)
   // Then, we load x16 from where the stack
   // pointer is pointing, but add 4 to the pointer
   lw x16, 4(sp)
   lw x15, 8(sp)
   lw x14, 12(sp)
   lw x13, 16(sp)
   lw x12, 20(sp)
   lw x11, 24(sp)
   // Then, we load x10 from where the stack
   // pointer is pointing, but add 28 to the pointer
   lw x10, 28(sp)
   // Now that all registers have been restored,
   // it is safe to move our stack pointer back up
   addi sp, sp, 32

   ret
```

Keep in mind that the stack pointer has to be **4-byte aligned**, which is why we are subtracting/adding 4 each time.

Since this alignment requirement is maintained at hardware level, if you do not follow this rule, you program will not proceed after the faulty instruction. Again, you will need to follow the LIFO rule of stack to load back the registers properly.

2. ASM with C

In this section, you will be writing an assembly program that relies on standard C library functions.

2.0 Autograder update

Since the recursion functions might be hard to debug, the autograder now will print the full inputs as well as the expected outputs for problem 2.2, 3.1, and 3.2.

However, as there are more texts to print, the target console might respond a bit slower (takes ~10 seconds to dump the strings to the console, but the actual code execution will finish in an instinct) and a bit difficult to locate the error testcases. Therefore you could now specify which problem you want to test with by substituting the `TEST_ALL` symbol under `test:` label with the predefined ones in the `lab3.S` template. ORing these test flags are supported so you could run any combination of the tests (e.g. `TEST_FIB | TEST_BSEARCH` will just run the last two problem tests).

Be sure to replace the test variable back to `TEST_ALL` when you finishes debugging each problem individually or else a warning will be printed at the end of test output stating not all testcases are run.

> If you run `TEST_ALL`, there will be a total of 120 testcases or 30 per problem.

2.1 Better call quicksort

In this problem you will write a function named `void asm_sort_int(int32_t* arr, uint32_t n)` that relies on `qsort` in C standard library to sort in ascending order. The C equivalent implementation is as follows:

```
void asm_sort_int(int* arr, uint32_t n) {
    // We sort array `arr` with `n` elements
    // Each element is of size 4 bytes
    // Using `asm_cmp` as the compare function
    //    You will need to load the memory address
    //    of asm_cmp when passing it.
    qsort(arr, n, 4, asm_cmp);
}

int asm_cmp(const void *a, const void *b) {
    // Compare function used by the qsort
    // You do not need to worry about typecasting in asm
    // just load them as signed words (32-bit)
    int tmp = *(int *)a - *(int *)b;
    if (tmp < 0)
        return -1;
    else
        return 1;
}
```

> Note: the function signature for `qsort` is:
>
> ```
> void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))
> ```

> Hint: you might want to consider the pseudo-assembly `la xn, LABEL` to load a label address into a register.

2.2 String concatenation

This problem will ask you to write a function named `char* asm_strconcat(char * str1, char * str2)`. This function will first allocate memory space with `malloc`, concatenating `str1` and `str2`, and return the resulted string. The C equivalent implementation is as follows:

```
char * asm_strconcat(char *str1, char *str2) {
    // Get string length
    int n1 = strlen(str1);
    int n2 = strlen(str2);

    // Assign space for the concatenated string
    // sizeof(char) = 1
    int size = n1 + n2 + 1;
    char * buf = malloc(sizeof(char) * size);

    // Use `memcpy` to copy string
    memcpy(buf, str1, n1);
    memcpy(buf + n1, str2, n2);

    // Write null char and return it
    buf[size - 1] = '\0';
    return buf;
}
```

> Note: `strlen`, `malloc`, and `memcpy` are all standard C library functions. You should check out their function signatures online to determine what arguments you need to set up for each of them.

3. Recursion

> Recursion: *noun*.
> Definition:
>   See *recursion*
>   – ECE 362 Dictionary

Recursion involves dividing a given problem into subproblems, solving them, and combining the solutions to get the overall solution for the problem.

When coding a recursion function in assembly, be aware that your function will become both the caller and the callee. Thus, you will need to consider saving registers of both types when entering a function, and when calling a function.

3.1 Fibonacci revisited

Let's redo what we have in lab 2 with Fibonacci but this time with recursion! You will implement the function `uint32_t asm_fib(uint32_t n)` which will accept an index term `n` and return the F_n fibonacci term (in particular: $F_0 = 0$, $F_1 = 1$). The C equivalent implementation is:

```
uint32_t asm_fib(uint32_t n) {
    if (n < 2) {
        // Base cases
        // F0 = 0
        // F1 = 1
        return n;
    } else {
        // Inefficient way (O(2^n) time complexity)
        // to generate the fibonacci series,
        // but good intro problem for recursion
        // Fn = Fn-1 + Fn-2
        // To see why it is inefficient, asm_fib(n-2)
        // just does again what asm_fib(n-1) does for the
        // n-2 terms
        return asm_fib(n - 1) + asm_fib(n - 2);
    }
}
```

3.2 Binary search in assembly

In this problem, you will implement a binary search algorithm in assembly. The binary search will search for an element in a sorted array with time complexity $O(\lg n)$. It will compare the middle element with the search key, and decide which subarray in which to continue the search. When the algorithm terminates, the program should either return the index of the first discovered element, or `-1` if the key cannot be found. A C implementation is provided below:

```
// Return the index of the element in the array
// if the element does not exist, return -1 instead
// arr: integer array to be searched
// key: the element we want to search on
// start: start index of subarray, inclusive
// end: end index of subarray, inclusive
//
// so the initial search of array
// A = {1, 2, 3, 4, 5} on key = 3
// will be
// asm_bsearch(A, 3, 0, 4);
int asm_bsearch(int *arr, int key,
                    int start,
                    int end) {
    if (start > end) {
        // We could not find the element
        return -1;
    } else {
        // Check the middle element
        // to decide which subarray should we
        // search on
        // div by 2 can be done by right shifting by 1
        uint32_t mid = (end + start) / 2;
        if (arr[mid] < key) {
            return asm_bsearch(arr, key, mid + 1, end);
        } else if (arr[mid] > key) {
            return asm_bsearch(arr, key, start, mid - 1);
        } else {
            return mid;
        }
    }
}
```