Jerry Xia, 260917329 | Lab 1 Report

Part 1.1

A brief description of each part completed (do not include the entire code in the body of the report).

- This is an iterative fibonacci program that uses registers, instead of the stack, to compute the *n*th fibonacci number.

The approach taken (e.g., using subroutines, stack, etc.).

- This program uses a variable, which we call n, and uses registers r0 - r3 to store values, essentially accumulating a sum iteratively and then returning it at the end within the LOOP function. There is one subroutine call at the beginning for the LOOP, so that we are able to 'bx lr' to this call once LOOP is finished. A counter was used to decrement the value by 1 each time, and this way when n-1 iterations occur, then the nth fibonacci is returned in register r1, is moved to r0 and then the function branches back to the beginning, where it then branches to the end, thus terminating the program.

The challenges faced, if any, and your solutions.

- The return value was originally not in r0. Moving it into r1 at the end of the program proved to work well.
- Not many challenges were faced in this part of the lab.

Possible improvement to the programs.

- To keep the registers 'clean' upon terminating the program, the LOOP should have pushed all registers onto the stack and then popped them off at the end. The only reason this wasn't pursued was because the registers were in use the entire time throughout the duration of the LOOP.

Part 1.2

A brief description of the part completed.

- This is a recursive fibonacci program that uses the call stack and a frame pointer to compute the *n*th fibonacci number.

The approach taken (e.g., using subroutines, stack, etc.).

- This program uses a variable, which we call n, and registers r0 - r3 to store values. All other values it pushes onto the stack.
- This program also uses subroutine calls so the call stack doesn't get corrupted.
- Upon entering fib, the frame pointer is pushed onto the stack, and the current value of the stack pointer is moved into the frame pointer (r11)'s current value. We then:
    - Load the value from r2, which is what we pass to the function, into r2 from the stack, comparing it to 0 and 1.
    - If the value is in fact 0 or 1, then return that value in register r0 for sake of convention.
    - If the value is greater than 1, then subtract 1 from the value, place it into r2. We will push this argument onto the stack and then call fib again recursively.
    - After this, move the stack pointer up to essentially (pop) a value. This deallocates the n - 1 variable we stored in the step above.
    - We then push r0 on the stack. This will be used later.
    - We then calculate the value for n - 2, store it onto the stack, and compute fib until the return value is once again placed in r0.
    - We shift the stack pointer up to discard the n - 2 value.
    - We pop n - 1 from the stack into r1.
    - We compute add r0, r1, r0.
    - We move the frame pointer into the stack pointer to return to our previous call, and (pop) the next value into the frame pointer for use later.
    - And then we branch back to our caller, which in our last case is _start, or in our recursive case is fib.

The challenges faced, if any, and your solutions.

- One main challenge was raised in this part of the lab. After the idea of a frame pointer was introduced in class, I began working on this. Using this idea, I had many problems with knowing when to push and pop the frame pointer into the stack pointer, and much more. I eventually identified the pattern of pushing the frame pointer onto the stack, moving the stack pointer's value into the frame pointer register, and once a subroutine was finished, it was necessary to move the frame pointer's value into the stack pointer and then pop the value of the old frame pointer into the register of the frame pointer r11. Again, this was only through repetition that I was able to figure this out.
- Despite this, there were no other significant issues - others such as branching and linking were solved rather quickly, along with loading into a register, other small tasks, etc.

Possible improvement to the program.

- I think the frame pointer is a good use of programming principles and methods.
- However, this program could have also been feasible without it. By using the call stack to push arguments and branching to certain cases, such as returning 0 or 1, could have worked. However, this method was used for the sake of reducing this redundancy.

Part 2

A brief description of the part completed.

- The original array was placed into the data section of the program, along with the kernel.
- A space of 400 bytes was allotted for the answer.
- All six variables (iw, ih, kw, etc.) were reserved space as well.
- It was found that by storing y as a word in memory, and loading it into the register each time was a feasible way of computing the logic of the program. This allowed space for r0 to be the sum that gets accumulated and eventually stored.

The approach taken (e.g., using subroutines, stack, etc.).

- This program uses a variable, which we call n, and uses registers r0 - r3 to store values, essentially accumulating a sum iteratively and then returning it at the end within the LOOP function. There is one subroutine call at the beginning for the LOOP, so that we are able to 'bx lr' to this call once LOOP is finished. A counter was used to increment the value by 1 each time, and this way when n-1 iterations occur, then the nth fibonacci is returned. The nth fibonacci number is stored in r1, is moved to r0 and then the function branches back to the beginning, where it then branches to the end, thus terminating the program.
-

The challenges faced, if any, and your solutions.

- Considering many operations were to be done, I had to use many registers and oftentimes the call stack was corrupted.
    - I kept track of what variables were pushed and popped off the stack more easily by pushing and popping immediately when the values were used. This way I could easily track what was on the stack and what was off.
- Many branch and link instructions were used. By acknowledging the callee-save convention and branch and link instructions the problems with branching were minimal.
- I also considered using multiplication rather than the lsl method learned in class. Using this made things less tedious.

Possible improvement to the programs.

- One flaw was that I ended up spending far too much time trying to fix a variable that was named incorrectly. After fixing this, the program was able to run flawlessly. However, I learned that I should, instead of cutting and pasting code for simplicity, just write it out again.
- On another note, loading y from memory was unnecessary. This part was only done due to the fact I had misplaced a variable for another, and my calculations were wrong - so I overhauled the logic of my program to fix this flaw, which wasn't ideal.
    - This would save the time performance of the program as y would simply be stored in one of the registers.

Part 3

A brief description of each part completed.

- This is a bubble sort program that takes in an array of integers, a size of the list, and sorts the list using a double nested for loop.

The approach taken (e.g., using subroutines, stack, etc.).

- There are two loops, one being labelled *step* and the other being labelled *i*.
- Everytime each label is called, a branch and link is used. Registers used are pushed onto the stack, return values or computed values (values that are only needed momentarily) are moved into registers r0 - r3, and then popped off the stack once they are finished with.
- The subroutine calling convention is preserved.

The challenges faced, if any, and your solutions.

- This was the first time that I had to implement subroutines. For a large part of the time, I wasn't able to get the branch and link to work, so I had to resort to a method that produced a large amount of errors. Only through repetition and strengthening my understanding of the stack, as seen in class, could I solve this problem.
- Using only the registers r0 - r3 proved to be a problem at first but I was able to push values onto the stack, compute what I needed to, and then pop them off after.
- Other than this, many challenges were faced in this part of the lab.

Possible improvement to the programs.

- To make the code more readable, perhaps I could have reset the counter i within the i block, and likewise with the step loop and the step counter.
- Branching and linking could potentially not be necessary, but I believe through this implementation a superior method to that was implemented.
    - This is because once a subroutine (in this case a loop) is terminated, the code reviewer can easily see what will execute next in the program. This is helpful in not only debugging, but also being comprehensive of the program itself and how it would work in C. As well, having a large amount of branches would be the only way to implement this without subroutines, which would be tedious and closer in reality to "spaghetti" or "unnecessary" code.