ECSE 324 - Lab 4 Report

Alexander Liu

April 12th, 2024

**Part 3: Performance Analysis A**

**Code Overview**

My subroutine *GoL\_draw\_grid\_ASM* leverages the *VGA\_draw\_line\_ASM* subroutine(which leverages *VGA\_draw\_point\_ASM*) to draw the game grid. *VGA\_draw\_line\_ASM* is strictly configured to draw horizontal and vertical lines only. The colour for the lines is passed to *GoL\_draw\_grid\_ASM* in A1. I start off by instantiating my coordinates (x1, y1) and (x2, y2) which will be subsequently passed to the line-drawing subroutine. Then, I have 2 loops in the body of my grid-drawing subroutine with the first loop to cover all vertical lines and the second loop to cover all horizontal lines. There is some setup required to transition from one loop to another which occurs upon exiting the first loop.

**Performance Analysis**

Before making any analysis on performance, it is important to establish definitions. In this report, a data memory access will refer to an explicit load or store instruction which excludes fetching instructions as memory accesses. We define data memory access this way because we are more concerned with the different subroutines’ runtimes. There are 3 subroutines that we must analyze to gauge performance: *GoL\_draw\_grid\_ASM*, *VGA\_draw\_line\_ASM* which is called inside the body of the encapsulating subroutine *GoL\_draw\_grid\_ASM*, and *VGA\_draw\_point\_ASM* which is called within *VGA\_draw\_line\_ASM*. The easiest way to account for the number of data memory accesses performed by all 3 subroutines to draw the grid is through breakpoints. The methodology employed here is to insert a breakpoint at the line where I call the subroutine *GoL\_draw\_grid\_ASM* and another breakpoint immediately after on the next line. The images below showcase where the breakpoints were inserted explicitly as well as provides the relevant data before and after the execution of *GoL\_draw\_grid\_ASM*.

A screenshot of a computer

Description automatically generated

**Figure 1.** Breakpoints inserted in the program’s disassembly

A screenshot of a computer

Description automatically generatedA white background with black text

Description automatically generated

**Figure 2.** Counter values before and after calling *GoL\_draw\_grid\_ASM* (respectively)

From figures 2 and 3, we can perform a subtraction between the number of data loads and sum the difference with the difference between the number of data stores. Doing this operation, we get [252102 - 230607] + [336019 - 307404] = 50110 data memory accesses in total. Moreover, the number of instructions executed by calling *GoL\_draw\_grid\_ASM* is 157522 and the runtime of the subroutine is 549 – 495 = 54 ms. This yields a data memory access to instruction executed ratio of 0.3181, meaning 31.81% of all instructions executed are data memory accesses. Overall, we can confidently say the most time is spent inside *VGA\_draw\_point\_ASM* since we need to call it to write to every pixel in every line in the grid. We have 15 vertical lines and 11 horizontal lines. Each vertical line has 240 pixels, and each horizontal line has 320 pixels. This means that we are calling *VGA\_draw\_point\_ASM* for a total of 7120 times while only calling *VGA\_draw\_line\_ASM* 26 times and *GoL\_draw\_grid\_ASM* once. Hence, the most total time spent would be within the *VGA\_draw\_point\_ASM* subroutine. If the time required to perform a data memory access were to increase, then we can expect the runtime of the subroutine to suffer consequently. More precisely, since 31.81% of instructions are data memory accesses, we can expect a somewhat substantial increase in runtime.

**Part 3: Performance Analysis B**

**Code Overview**

The *\_start* block of my program contains all the relevant setup required to initiate the game. The main block we are concerned with is *IDLE* where most of the user interaction and game state updates happen. My approach to registering keypresses is interrupt-based. However, *IDLE* polls the global *DATA* variable which holds the most recently fetched keyboard signal from the keyboard’s data register. It tries to match the *DATA* variable against relevant make signals corresponding to the keys that are relevant in game logic and performs the appropriate updates to the game’s state.

**Performance Analysis**

I will consider all instructions executed in *SERVICE\_IRQ* and *PS2\_ISR* as instructions attributed to I/O interactions or interrupts, and all instructions executed in *IDLE* to be instructions servicing user code or game logic. I am not attributing any instructions to polling as there is no consistent time frame between keystrokes. Therefore, code spent polling the *DATA* variable after a keystroke interrupt will be attributed to game logic as well. To get an idea of how many instructions and data memory accesses are executed in each category, I will go through two scenarios of updating the game board’s state: from pressing the N key to seeing the updated game board and from pressing D to seeing the updated cursor. My methodology in measuring the relevant data will be to insert a breakpoint at the beginning and end of *SERVICE\_IRQ* to measure the number of data memory accesses and executed instructions serviced to interrupts. I will also insert breakpoints at the beginning and end of the *update\_GoL\_board* and *move\_cursor\_right* subroutines to measure how many executed instructions and memory accesses are serviced to game logic in each case. With the way my ISR is set up, I will have to run measurements on independent runs. This is because my ISR empties the data register of the PS/2 keyboard, so once I reach the ISR’s first breakpoint, the ISR will empty the make and break signal of the keypress (unless I hold the key until I exit the ISR, but that would be impractical) which won’t match any of the conditions specified in the *IDLE* block.

A white background with black text

Description automatically generatedA screenshot of a computer

Description automatically generated

**Figure 1.** Counter values before and after the interrupt service routine

A white background with black text

Description automatically generatedA screenshot of a computer

Description automatically generated

**Figure 2.** Counter values before and after calling *update\_GoL\_board*

*A white background with black text

Description automatically generatedA white background with black text

Description automatically generated*

**Figure 3.** Counter values before and after calling *move\_cursor\_right*

From Figure 1, we can tell that running the ISR executes 830 instructions, performs 298 data memory accesses, and requires 1 ms of simulator runtime. Similarly, the *update\_GoL\_board* subroutine (Figure 2) executes 1704950 instructions, performs 535066 data memory accesses, and requires 496 ms of simulator runtime. Lastly, the *move\_cursor\_right* subroutine executes a total of 10464 instructions, performs 3360 data memory accesses, and requires 7 ms of simulator runtime. From these results, we realize that the bulk of the game logic is concentrated in *update\_GoL\_board* as it executes a total number of instructions and memory accesses that are orders of magnitudes greater than the ISR and moving the cursor (we can generalize that moving the cursor in all different directions will yield similar results to *move\_cursor\_right*). However, the takeaway here is that the ISR takes up minimal resources in terms of runtime and processing which is ideal as we would like to minimize the time servicing interrupts and maximize the time the processor spends on user code or game logic. From the gathered results, we can conclude that the time spent in the ISR is trivial and that the overwhelming majority of the program’s runtime and resources will be spent processing game logic.