Lab 4: Paint application (CSE12-02 Spring '23)

Due Friday, June 9th, 2023, 11:59 PM

Minimum Submission Requirements

- Ensure that your Lab4 Gradescope submission contains the following files (note the capitalization convention):
  - lab4_part1.asm
  - lab4_part2.asm
- Ensure that you pre-submit your answers so that you will know your final actual grade ahead of time. There are plenty of possible submission errors that may result in a bad score, even when your program appears to work. You can submit your code as many times as you want without penalty.

Brief Summary of Lab 4

This lab will develop a mini "Paint" application running on our emulated RISC-V computer. This application will take user input from the keyboard to draw patterns over a Bitmap display. A bitmap essentially "maps" a bit pattern (in our case a 32-bit/64-bit value) to a specific color for display. The address where we store this color value essentially behaves as a "pixel" when we output that value over a display unit.

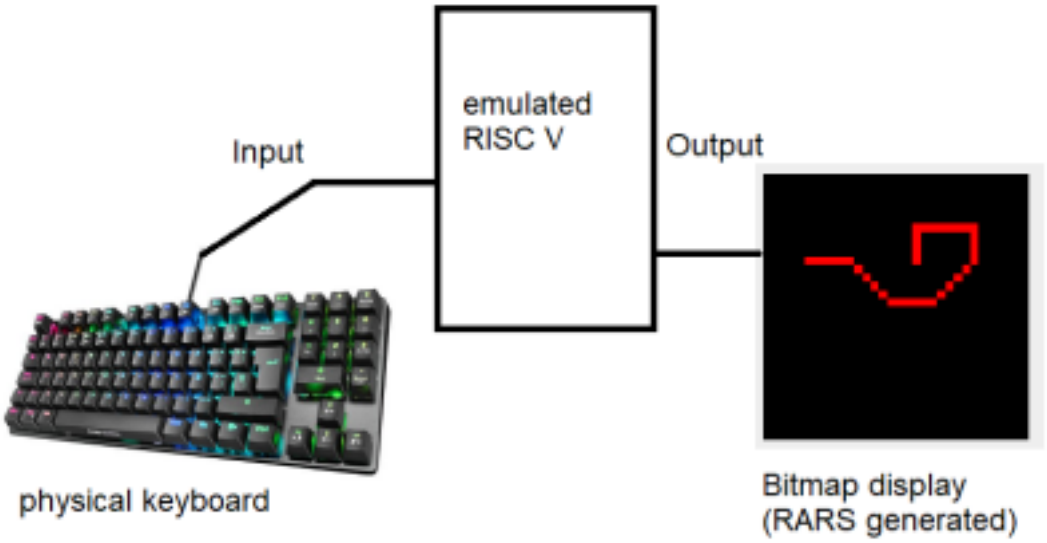


Figure 1 High level Lab4 visualization

Our given Paint application will draw against a black background. The user will specify a starting pixel coordinate. Then using the w, a, s, d keys, they can draw straight lines in left, right,up, down directions. By pressing q, c, e, z keys, they can draw diagonally as well.

As you read along in this document, you will be given hints so as to guide you in finishing Lab4. **So, you**

MUST carefully read every section in this document in order without skipping any! This lab is a lot easier than it seems. But you have to make sure that you properly set up the keyboard, and the display tool as described, in order to get it to work. Again, allow plenty of time for this lab (start early) and read the description carefully.

Resources

In the Lab4 folder in the course Google Slide, you will see the following files. They are meant to read (and understood) in sequence:

1. *add_function.asm* – This program accepts two integers as user inputs and prints their addition result. The actual addition is done through calling a function, sum. Sum accepts two arguments in the a0, a1 registers and returns a0+a1 in a0 register

2. ***(optional)*** *multiply_function.asm* – This program accepts two integers as user inputs and prints their multiplication result. The actual multiplication is done through calling a function, multiply. Multiply accepts two arguments in the a0, a1 registers and returns a0*a1 in a0 register. This function in turn calls the function sum, described previously, to do a particular addition. Thus, multiply function is an example of a ***nested function call,*** a function which itself calls another function, sum in our case.

3. *RISC-V_function_convention.pdf* – This pdf describes the conventions of how we use the various 32 RV64I registers in terms of creating a program with multiple functions. The documentation is provided such that students can start working on implementing the required function for Lab4_part1.asm. Studying the comments in add_function.asm alongside reading this pdf should be sufficient to create the required function for Lab4_part1.asm. The topic involves what registers to use and how and when to spill them (save them on the stack) and when and where to restore them. The decisions to be made are dependent on whether you are caller, a callee, (or both) and or a leaf function (a type of callee). The class slides on function calls, also addresses this subject with perhaps a little greater clarity than this PDF; so please review them.

4. *bitmap.asm* – This program generates a 32 by 32-pixel grid on the Bitmap display tool provided by RARS. Certain pixels are colored red, green and blue.

5. *lab4_part1.asm, lab4_part2.asm* – These are the 2 files you will need to modify and submit as part of Minimum Submission Requirements.

6. ***lab4.asm*** – This is the testbench file that will implement our Paint application if *lab4_part1.asm* and *lab4_part2.asm* are implemented correctly. These two student files are stitched into this .asm source code. **Thus, you should only read the file *lab4.asm* and NOT change or modify it in any form.**

Please download the .asm files and **make sure** to open them **in RARS Text editor only.** Else the comments and other important code sections won't be properly highlighted and can be a hindrance to learning assembly language intuitively.

Understanding the RARS bitmap display tool

At this point, you should be somewhat comfortable with writing simple leaf functions in assembly. Remember to approach tutors/TAs/instructors if you still feel the need to clarify some questions you might have regarding this aspect of the lab.

HTML color codes

For displaying various colors, the RARS bitmap tool uses the HTML color codes. In the link provided, note that the color code hex value is preceded by "#" instead of "0x". Both mean the same thing. For example, #FF0000 and 0xFF0000 both refer to the color **red.**

Running bitmap.asm on RARS

The following instructions would apply to any asm source code that attempts to use the RARS bitmap tool.

Open bitmap.asm on RARS. Now go to Tools->Bitmap Display. **You MUST apply the following settings as shown in the screenshot below.**

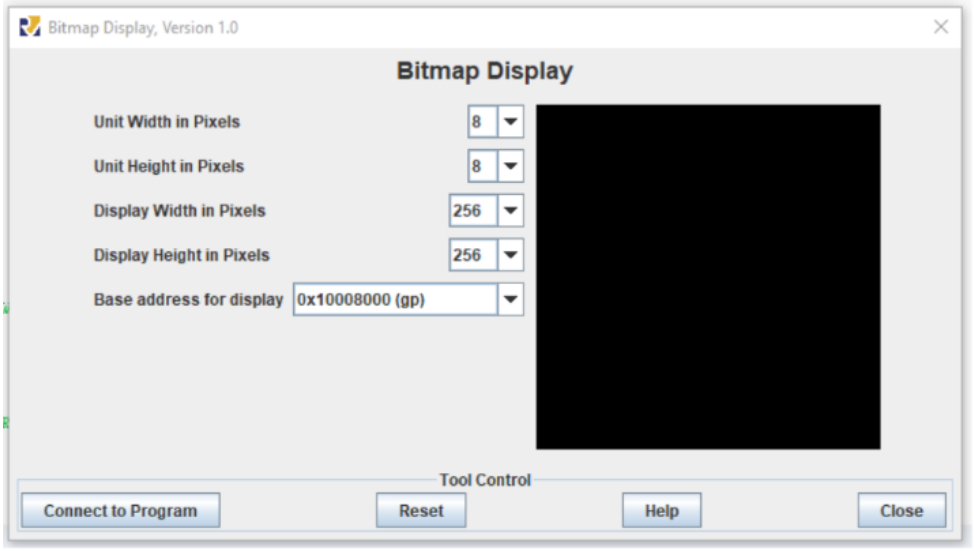


Figure 2 Required Bitmap Display settings for Lab 4

Now, click on "Connect to Program". Now on the main RARS window, click Assemble->Run. You should see the Bitmap display light up as shown below.

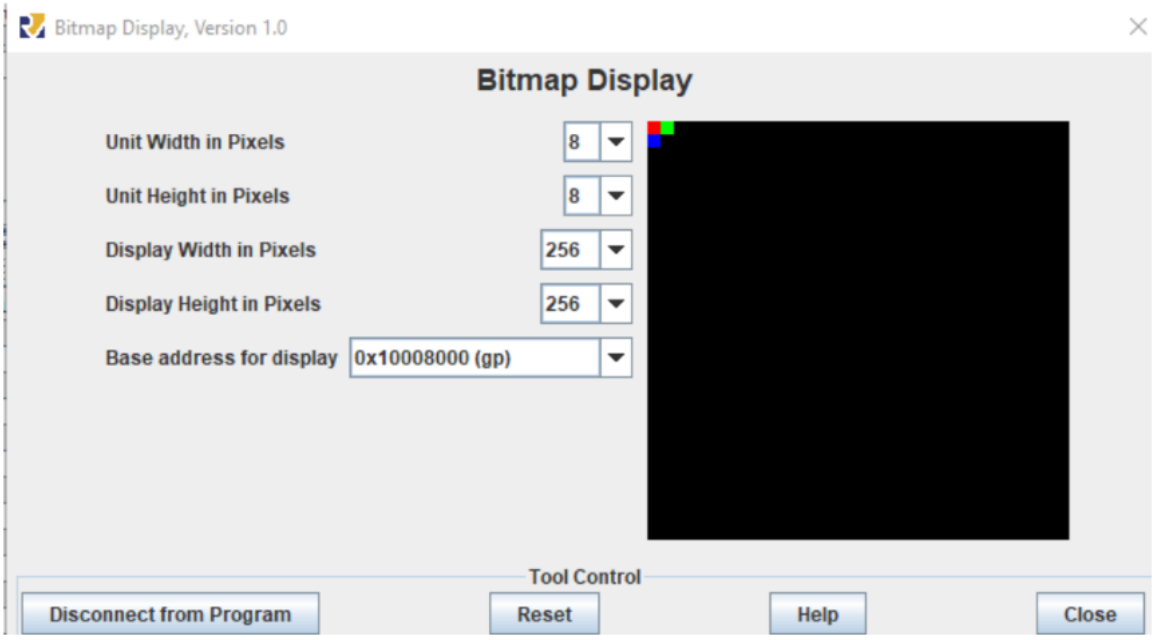


Figure 3 Bitmap Display after running bitmap.asm

So, what happened? Well, first off, the global pointer, gp, is a special register that points to the beginning of a certain area within a memory segment. In this case the .text section of the RISC-V memory model.

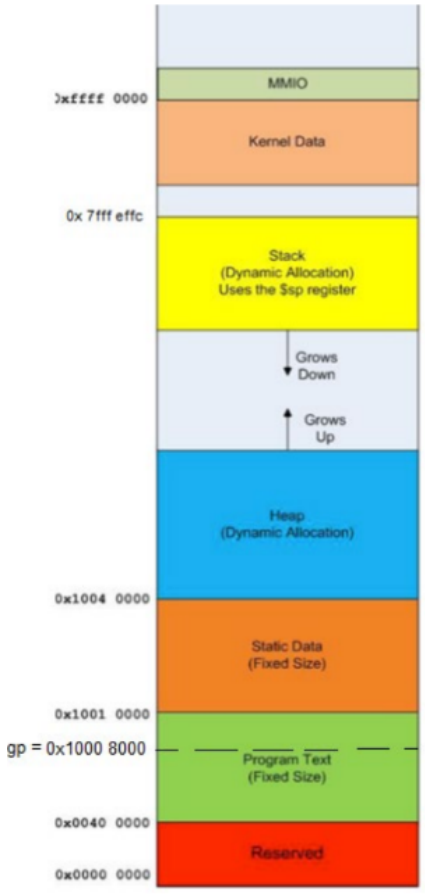


Figure 4 Memory location pointed to by gp register

The gp allows a compiler to globally access static data variables (global) relative to the base position indicated by gp. Here that is the address 0x1000 8000 in RISC-V. In practice the location of gp gets determined by your language toolchain, e.g. compilers, linker, etc. to match the system you have.

In the RARS Bitmap Display, when we select gp as the "Base address for display", it visually shows the memory contents of a 4 KiB region in main memory **starting from address 0x1000 8000.**

When we start the program, this 4 KiB region has all the bytes initialized to 0x00, which is the HTML color code for black. **Hence the black screen.**

Let us now do some math. First off, 4KiB = 4 * 1024 B = 2^{12} B [1 KiB = 1024 B]. In the *bitmap.asm* code, note that we are using a word data size to represent each of our pixels. That is, 4B. So, if we are to envision the Bitmap display as a square grid, then the linear dimensions of this grid are sqrt (2^{12} / 4) =32 units each. So, it is a 32 x 32 grid. Note from the Bitmap display showing the colored pixels in bitmap.asm that this grid is not the usual Cartesian grid. That is, the origin (0,0) does not start at the bottom left, but rather at the top left.

Lab 4 part 1

In part 1, you will need to modify the provided file, *lab4_part1.asm*. The file contains a function label, *xyCoordinatesToAddress*, which you need to write code for.

This function takes as input, an (x,y) coordinate of a Bitmap pixel, as defined in the previous section, as well as the base address for display, and converts it to the actual main memory address.
For example, when comparing to Figure 3:

xyCoordinatesToAddress ((0,0), 0x 1000 8000) = 0x 1000 8000. This pixel is shown as red.

xyCoordinatesToAddress ((1,0), 0x 1000 8000) = 0x 1000 8004. This pixel is shown as green.

xyCoordinatesToAddress ((0,1), 0x 1000 8000) = 0x 1000 8080. This pixel is shown as blue.

The (x,y) coordinates are all positive integers. As shown above, when we increase the x value, the pixel moves rightwards. When we increase the y value, the pixel moves downwards.

For implementing this function, you do not need to do any bounds check to see whether the arguments fall within any permitted range or not. Your function can be a direct linear algebraic function of the form ax+by+c.

NOTE: You CANNOT use a multiply RISC V instruction directly offered by the RARS emulator, e.g. as shown in Figure 4.2 below!

If you are observant enough, you will realize you do not even need to invoke the multiply function (as shown in multiply_function.asm for example) . As an engineer, think of the shortest most efficient simple set of RISCV instructions that achieves the same set of objectives.

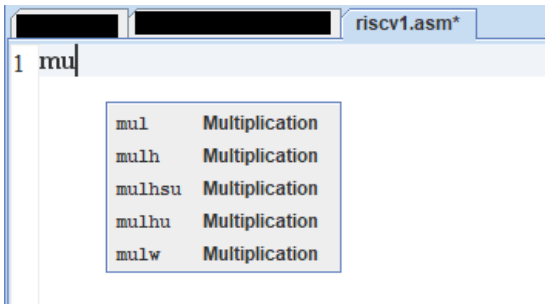


Figure 4.2 auto prompts for different multiply instructions offered on RARS. ***You are NOT allowed to use them in your code!***

Make sure to thoroughly read the comment in the *lab4_part1.asm* file to find out which registers are being used to pass arguments and to return a value to the caller**. Do not just assume anything!!** Ask a tutor/TA for help if you are confused by the directions given in the file!

Lab 4 part 2 and the Bigger Picture, so to speak

A demo of the completed Lab 4

You are now going to be shown what the completed Lab4 project looks like. This will give you perspective on how to implement *lab4_part2.asm*. **Make sure to place your completed *lab4_part1.asm* and *lab4_part2.asm* files in the same folder along with the provided *lab4.asm*, as well as the .jar RARS application which you will be using.**

If you coded part 1 and part 2 in this lab correctly, this is how the Paint application will run. First, connect the Bitmap Display tool as mentioned in a previous section. Also, open the "Keyboard and Display MMIO Simulator" from Tools. Select "Connect to Program" on the new pop up window that appears as shown below.

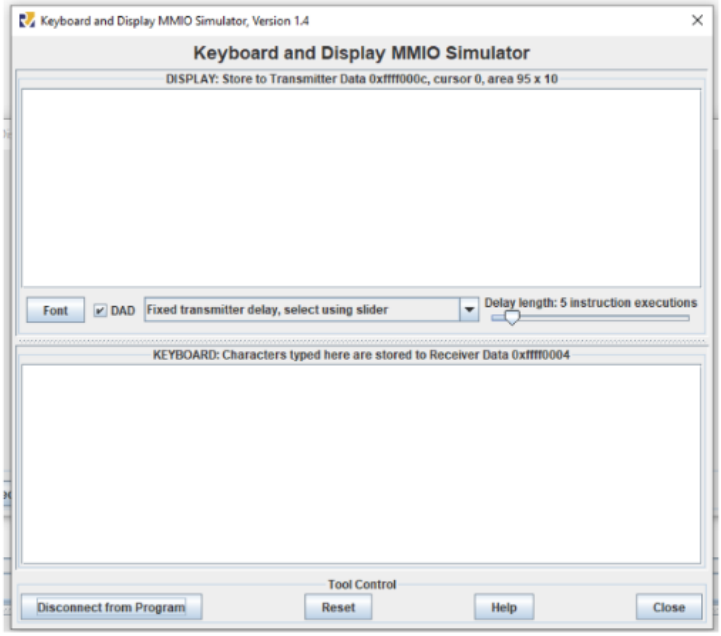


Figure 5 Keyboard and Display MMIO Simulator window

After lab4.asm is assembled and we click on Run, the I/O console will ask us for a starting coordinate on the Bitmap Display. We provide the coordinates (15,15). A red pixel appears at those coordinates on the Bitmap display and at the same time, we are told how to start drawing on the Bitmap Display with the keyboard. The corresponding displays are shown below.

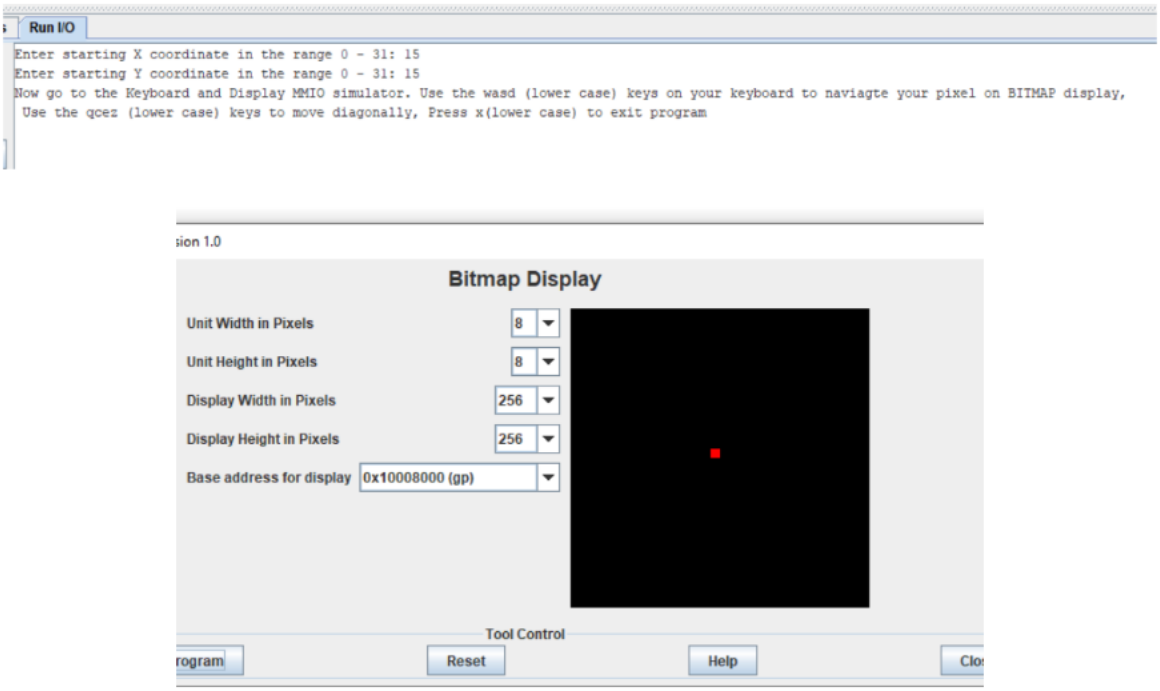
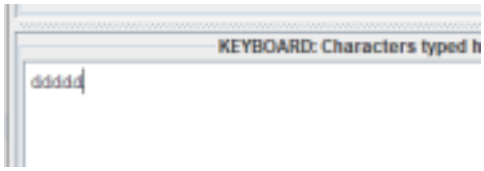


Figure 6 Starting pixel coordinates at (15,15)

Now, let us start drawing from the (15,15) starting position. On the Keyboard simulator, press d 5 times to move right 5 times as shown below.



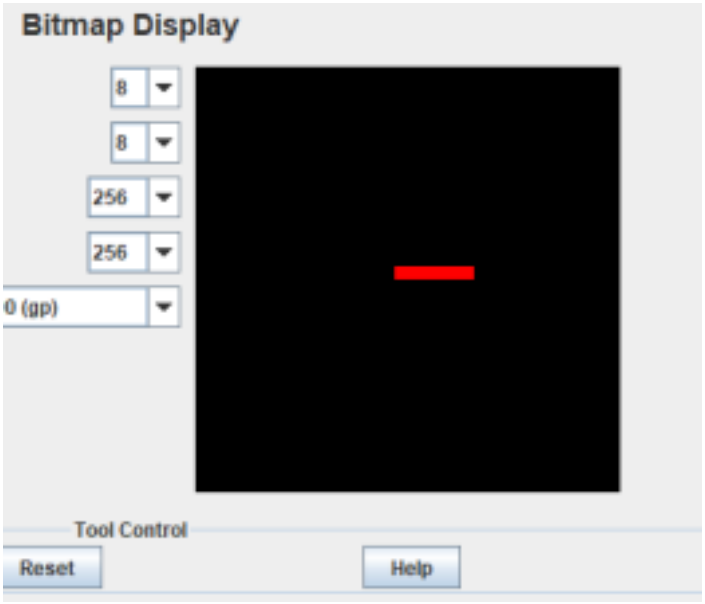


Figure 7 Moving rightwards

Now move downwards by 6 pixels.

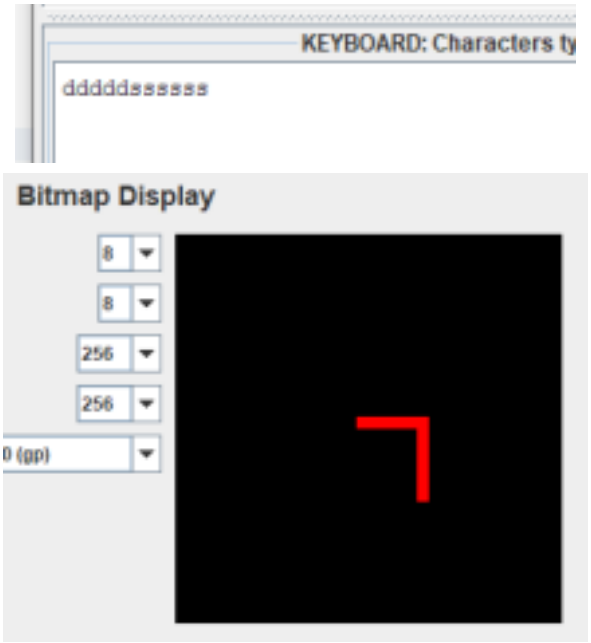
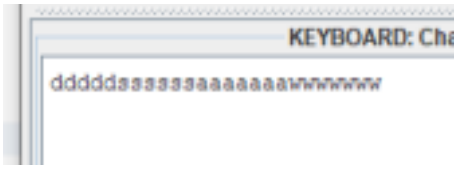


Figure 8 Moving downwards

Go left 7 pixels and then go up 7 pixels



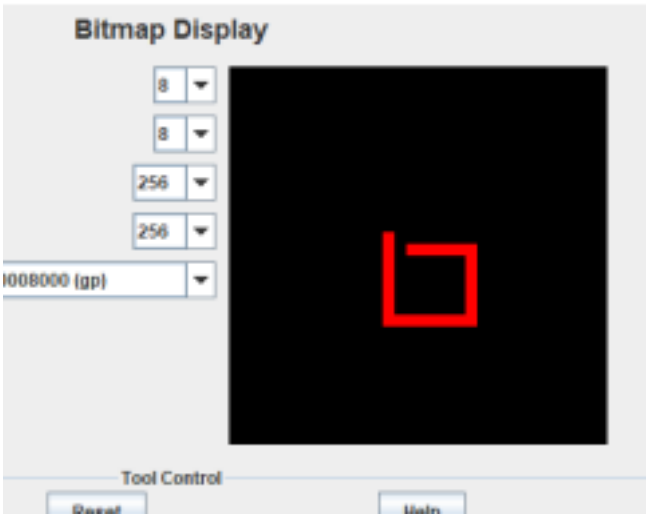


Figure 9 Moving leftwards and upwards

So wasd works. Now let's try the diagonals. Hitting the diagonal keys in the following order should give you the resulting picture on the Bitmap Display.

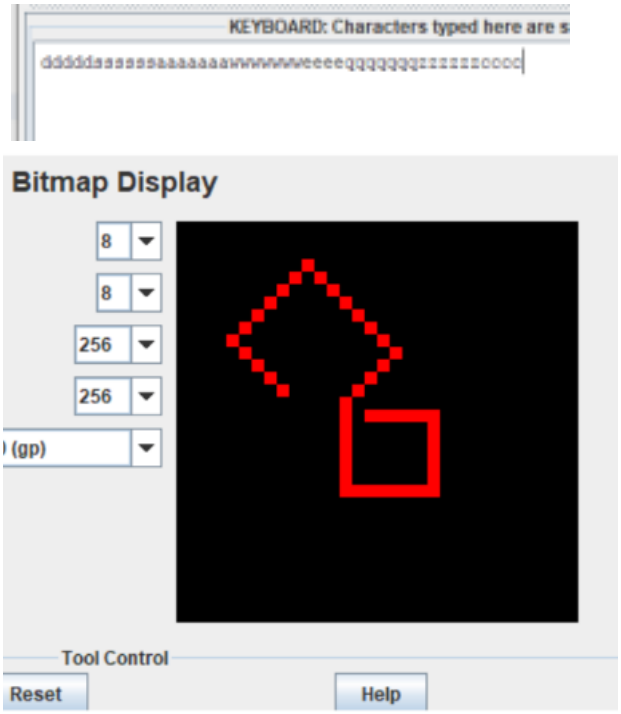


Figure 10 Moving diagonally

And we have successfully shown a demo of how our Paint application developed in RARS works! Sure, it's nothing fancy like MS paint or Photoshop but it can still do some rudimentary interesting artwork as shown below.

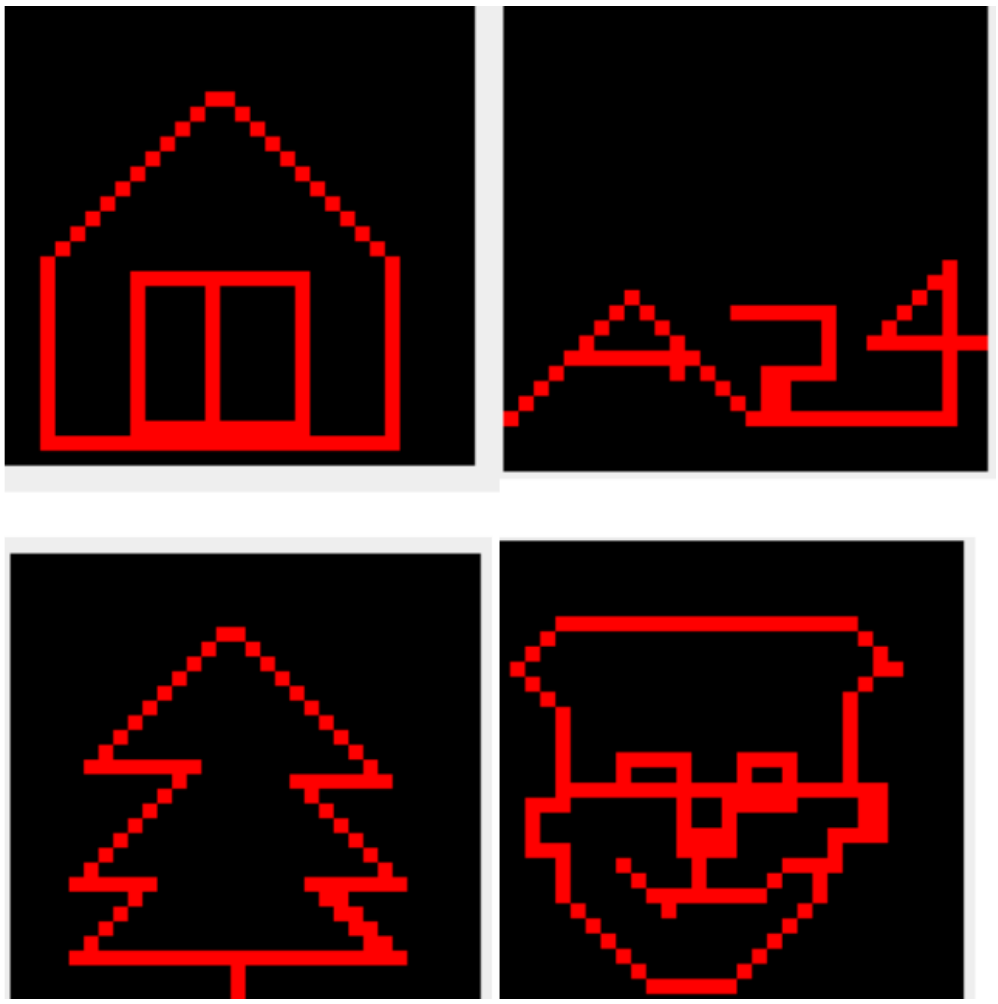


Figure 11 Works of Art (from top left): house, A24 logo, tree, a handsome boy

And that's how our Paint application works to create amazing works of art. Once you have completed both part 1 and part 2 , feel free to create your own!

Finishing part 2 of lab

At this point, you should have thoroughly read the *lab4.asm* file after opening it in the RARS text editor since it gives you important context on how to finish editing the file *lab4_part2.asm.*

At the line of code in *lab 4.asm* where we stitch in *lab4_part2.asm* using the include directive, the registers a0, a1 contain the x and y coordinates of the pixel respectively. Based on which label we jump to after receiving a user keyboard input (moveleft*, movedown, moveright, moveup, moveDiagonalLeftUp, moveDiagonalLeftDown, moveDiagonalRightUp, moveDiagonalRightDown*), you need to accordingly modify the a0, a1 values within each label description in *lab4_part2.asm*

HINT: After accordingly modifying the a0, a1 values within each of the aforementioned labels, make sure to jump back to a specific label in the original *lab4.asm* file! If you have read the lab4.asm file thoroughly (along with the given comments), you should be able to realize which label in lab4.asm you need to jump to from the labels in *lab4_part2.asm. This label should not be located far from where we stitch in lab4_part2.asm using the* ***.include*** *directive.*

Regarding the polling function in lab4.asm

In the *lab4.asm* file, you might have noticed the polling function having the *polling* label. This function is used to obtain the user input data when they hit on a key on the keyboard. The MMIO setup directly enters the user input and stores it into the memory location 0xffff0004.It indicates that this is a valid keyboard input by setting the memory location 0xffff0000 value to 1. Else, it stays at 0 value. In the polling function, we wait for the value at 0xffff0000 location to set to 1 so that we can pass on the user input stored in 0xffff0004 to the main function. Before we return to main, we make sure to reset the value in 0xffff0000 to 0.

Finishing up Lab 4

Automation

Note that our grading script is automated, so **it is imperative that your program's output matches the specification exactly**. The output that deviates from the spec will cause point deduction.

lab4_part1.asm, lab4_part2.asm

These files contain your pseudocode and assembly code. You do not need to insert any header comment in this code. But make sure to provide relevant comments both for your convenience as well as for the human grader in case there needs to be a manual grading.

A Note About Academic Integrity

This is the lab assignment where most students start to get flagged for cheating. Please review the pamphlet on Academic Dishonesty and look at the examples mentioned there for acceptable and unacceptable collaboration.

You should be doing this assignment completely by yourself!

Grading Rubric (100 points total)

The following rubric applies provided you have fulfilled all criteria in Minimum Submission Requirements.

30 pt *lab4_part1.asm, lab4_part2.asm* assemble without errors.

30 pt *lab4_part1.asm* provides correct output. No native multiply instruction from RISCV allowed!

40 pt *lab4_part2.asm* provides correct output.

- 5 pt w key entry works
- 5 pt a key entry works
- 5 pt s key entry works
- 5 pt d key entry works
- 5 pt q key entry works
- 5 pt z key entry works
- 5 pt e key entry works
- 5 pt c key entry works