← Lab 6: Graphics and Input

Finish by midnight on Sunday, 10/28

> Please download the newest MARS, `MARS_2191_c.jar`, from **the software page**.

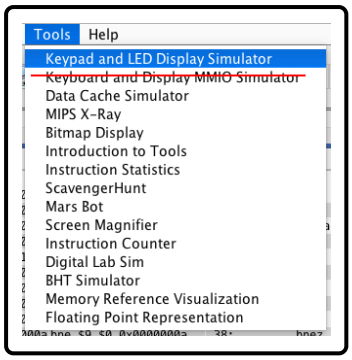
In this lab, you'll start using the **Keypad and LED Display Simulator** plugin to draw some sweet low-res graphics!

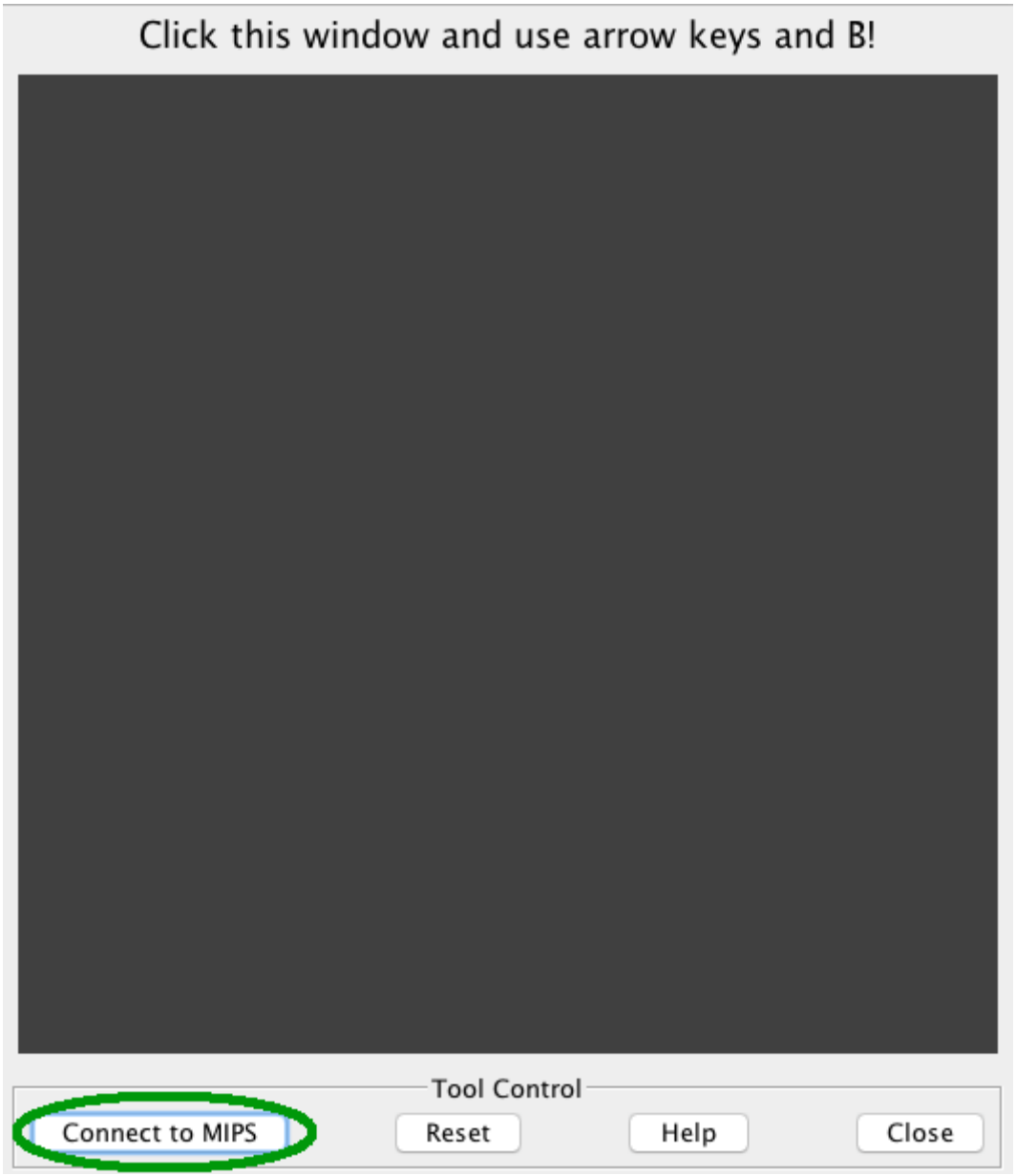
Make a file named `abc123_lab6.asm` in a new directory.

Opening the plugin

In MARS, go to to **Tools > Keypad and LED Display Simulator.** *Not Keyboard and Display MMIO Simulator.*

This will pop up a window. Click the "Connect to MIPS" button in the bottom left.





Once it's connected, **you don't have to close the window or reconnect it.**

You can re-assemble and re-run your program as many times as you want while the display is open.

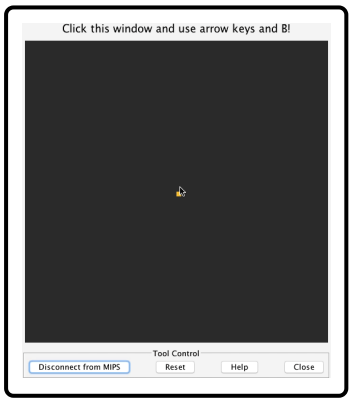
Using the plugin from your code

- **Right-click this link and "save link" or "download link".**
  - **When you save it, make sure it's really named `led_keypad.asm` and not `led_keypad.asm.txt`.**
- Put it in the same directory as your `abc123_lab6.asm` file.
- Then, in your `abc123_lab6.asm` file, add this at the **very top of the file:**

```
.include "led_keypad.asm"
```

Now you can use the constants and call the functions from `led_keypad.asm`!

Okay the lab for real now



For your second project, you'll make an interactive video game. Today's lab has all the same parts as a game, but very simplified. It will look something like the thing on the right when you're done.

The way *any* interactive program works is like this:

1. **wait** for a little while
2. check for user **inputs**
3. respond to those inputs by updating your program **state**
    - "state" means "your variables, data structures, etc."
4. change the **output** (screen) to reflect the new state
5. loop back to step 1

You had a similar program flow in project 1: user's turn, get inputs, dealer's turn, update variables, loop again. The difference is... it's *faster*. The player and opponent(s) get **60 turns every second!** AAAAAH!

---

1. First steps

Making your state variables

- Make two global variables (in `.data`) to hold the **dot's X and Y coordinates.**
- Name them appropriately.
- They should be words.
- **Initialize them both to 32.**

Making the main loop

The best place for the main loop is - you guessed it - in `main`.

- Make a `main` function like you have before.
    - Don't forget `.globl main.`
- Inside `main`, make an **infinite loop.**
    - This is the loop we talked about a little further up.

Waiting a little while

This is important. Without it, your program will run WAY too fast and you might not be able to stop it without force-closing MARS.

Syscall **32** lets you pause your program for a bit. It takes the number of **milliseconds** to wait in `a0`. There are 1000 milliseconds in one second. You want this loop to run **60 times per second.** So *how many milliseconds* do you have to wait? Do this syscall as the **first thing** inside your main loop.

> If MARS stops responding when you assemble your program, you didn't download the newest version of MARS.

---

2. Drawing the dot on the screen

- **Make a new function** called `draw_dot`. It will have no arguments and return nothing.
- **Call it** from your main loop *after* the waiting code.

> The screen is a 64 x 64 pixel display. Each pixel is a colored square that can be one of eight colors: black, red, orange, yellow, green, blue, magenta, or white. The `led_keypad.asm` file gives you many functions for drawing things onto the screen. The one you're about to use sets one pixel to a color.

It should do the equivalent of this code:

```
display_set_pixel(dot_x, dot_y, COLOR_WHATEVER);
```

- `display_set_pixel` is a function I gave you in `led_keypad.asm`. *You do not have to write it!*
- `dot_x` and `dot_y` are your state variables.
- `COLOR_WHATEVER` is whatever color you want to use.
  - Except black. You can't see a black dot on a black background. 😉
  - Look at the top of `led_keypad.asm` for the constant names.
  - **Use the named constant, not a number!** That's why we name constants!

Now if you run it, you should see….. nothing!!! What???

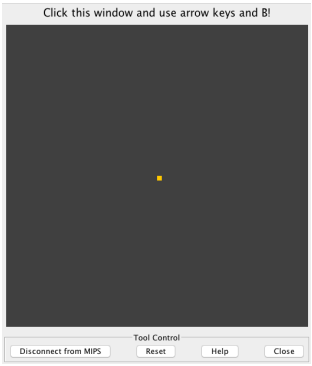
---

3. Making your drawings appear

In your main loop, after calling `draw_dot`, call `display_update_and_clear`. NOW your dot will appear.

What's going on?

The display plugin is *double-buffered*. When you draw to the screen, you are really drawing to a hidden area in memory first. Then you must call a function to actually *show the graphics on the screen*. This technique avoids problems when the screen is redrawn while you're in the middle of drawing things, causing weird graphical artifacts.



Like this.

> You should only call `display_update_and_clear` ONCE per main loop iteration, after drawing everything.

---

4. Moving it around

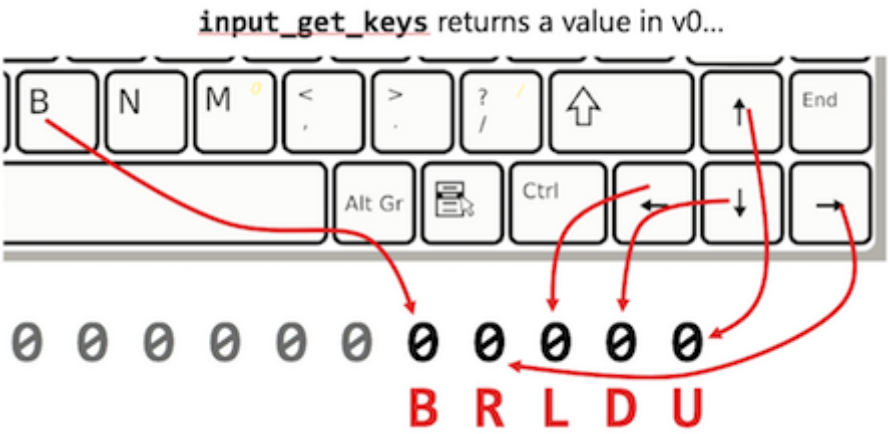
- **Make a new function** called `check_for_input`. It will have no arguments and return nothing.
- **Call it** from your main loop, **after waiting** but **before drawing**.

The way input works in this plugin is that you use the **arrow keys and B key on your keyboard.** Then, your program can detect that by using `input_get_keys`, another function from `led_keypad.asm`. It returns which keys are being held down, but it does so by returning **bitflags.**

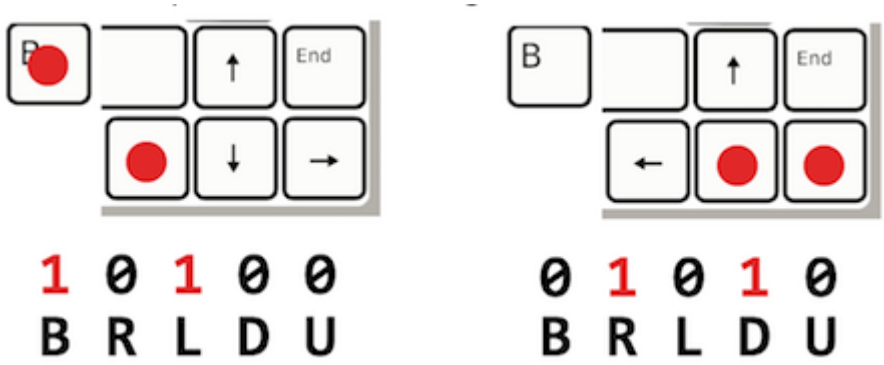
Bitflags

A special case of bitfields is when all the fields are **1 bit long.** In that way, we can think of an integer as a small array of boolean values. We call this **bitflags.**

`input_get_keys` returns an integer where the lower 5 bits represent the four arrow keys and the `B` key.



OK, tangent over



Here's pseudocode for what your `check_for_input` function should do. This is **not** an if-else if-else if... This is a sequence of 4 separate *if*s.

```
v0 = input_get_keys()

if((v0 & KEY_L) != 0) // bitwise AND!
    dot_x--;
if((v0 & KEY_R) != 0) // KEY_L, KEY_R etc.
    dot_x++;
if((v0 & KEY_U) != 0) // use the constant r
    dot_y--;
if((v0 & KEY_D) != 0) // don't hardcode the
    dot_y++;
```

*You might be wondering why we **decrease** the Y coordinate when pressing up. It's because the origin (0,0) is at the **top-left** of the screen, and the Y axis increases **downwards.***

Now run your program, **click the display,** and use the arrow keys on your keyboard. It should work, but... **try going off the top or bottom of the screen. What happens?**

---

5. Moving it around, *without* crashing

What's happening is your x and y coordinates are going negative or off the sides of the screen. On the top side, you're going to start **writing into a part of memory you're not allowed to, so it crashes.**

To prevent this, you have to limit the x and y coordinates.

At the end of your `check_for_input` function, before it returns, do the equivalent of the following:

```
dot_x = dot_x & 63; // bitwise AND!
dot_y = dot_y & 63;
```

Remember what this does? We learned about this as a shortcut for another mathematical operation...

Now your dot should wrap around to the other side like in the gif at the top of this page.

Fun things to try

- Try changing `jal display_update_and_clear` to `jal display_update`. Wheee!
- Instead of drawing the dot with a constant color, use a variable to hold the color, and have the B key change the color.
- Play around `display_fill_rect` in `led_keypad.asm`. The comments document its arguments and behavior.

Submitting

> Do not submit `led_keypad.asm`. Just your lab file, thanks.

Make sure your file is named `username_lab6.asm`, like `jfb42_lab6.asm`.

Submit here.

Drag your asm file into your browser to upload. **If you can see your file, you uploaded it correctly!**

You can also re-upload if you made a mistake and need to fix it.