CSC258 Assembly Final Project: Frogger

Due dates:

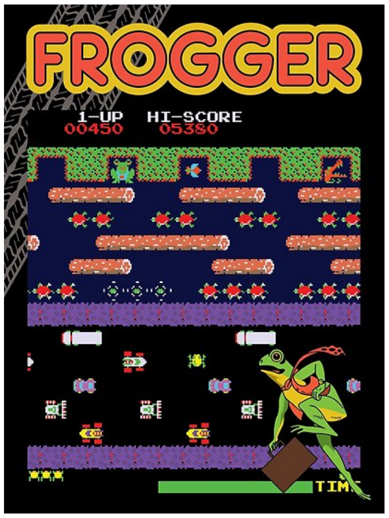
- Final Demo: Tuesday Dec 7 & Wednesday Dec 8, 6pm-9:20pm.
- Milestone 3 Demo: Tuesday Nov 30 & Wednesday Dec 1, 6pm-9:20pm

  This project must be completed individually, with demos performed in your lab time.

Document Updates

- **Nov 8, 2021**: Uploaded original project handout.
- Nov 13, 2021: Updated due dates
- Nov 18, 2021: Updated some silly typos
- Nov 29, 2021: Added some proposed features.

Overview



In this project, we will implement the popular arcade game Frogger using MIPS assembly. If you're not familiar with this game, you can try it out here.

Since we don't have access to physical computers with MIPS processors (see MIPS guide here), we will test our implementation in a simulated environment within the MARS application, which simulates the bitmap display and keyboard input of a MIPS processor.

How to Play Frogger

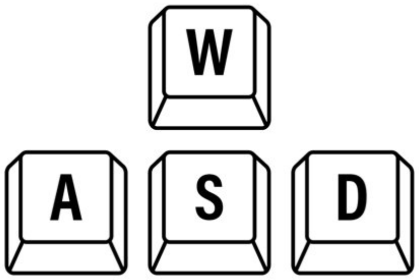
The goal of this game is to get a frog from the bottom of the screen to one of the safe areas at the top. The game is presented in a top-down perspective where the lower half of the play area is a road where cars and trucks move horizontally across the screen (which you need to avoid). In the upper half of the play area is a water area with floating turtles and logs that also travel horizontally across the screen. Unlike the cars and trucks, the frog needs to jump on these to reach the top of the screen.

If the frog collides with a vehicle at the bottom or falls into the water at the top, you lose a life. The game ends when you lose three lives or when all five safe areas at the top are filled.

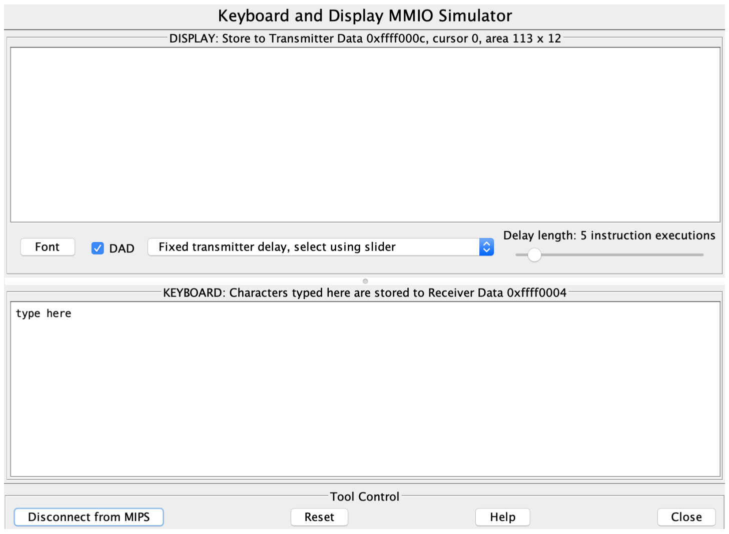
Game Controls

This game uses the WASD input keys to control the frog through the play area:

- The "w" key makes the frog move to up,
- The "a" key makes the frog move to the left.
- The "s" key makes the frog move to down,
- The "d" key makes the frog move to the right.



This project will use the Keyboard and MMIO Simulator to take in these keyboard inputs. In addition to the keys listed above, it can also be helpful to have a key such as "s" to start and restart the game as needed.



Keyboard and Display MMIO Simulator

DISPLAY: Store to Transmitter Data 0xffff000c, cursor 0, area 113 x 12

Fixed transmitter delay, select using slider

Delay length: 5 instruction executions

KEYBOARD: Characters typed here are stored to Receiver Data 0xffff0004

```
type here
```

When no key is pressed, the frog sits at its current position on the screen. If pressing a key moves the frog into a vehicle in the lower half of the screen or into the water in the upper half, the frog is reset back to the safe row at the bottom.

Project Goals

This handout describes the basic contents and behaviours of the game. The project will be assessed by the TAs in your weekly demo times, similar to the labs. If you implement and demo something similar to what we specify below, you'll get project marks for each part that you demo. The final component needed to receive full marks for the project is to implement additional features that bring your game closer to the actual arcade game. More details on this below.

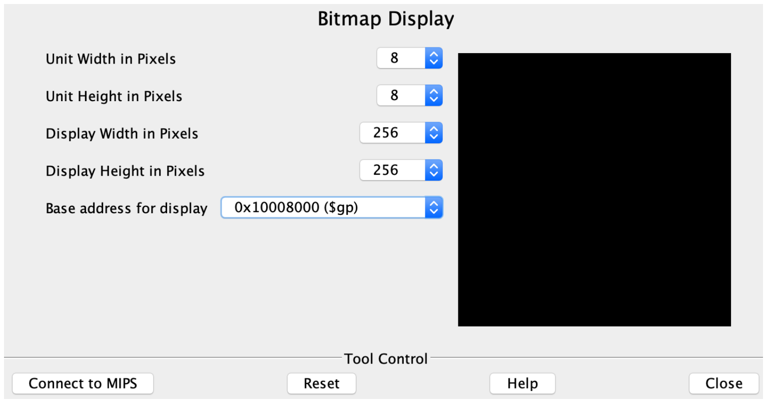
Technical Background

You will create this game using the MIPS assembly language taught in class. However, there are three concepts that we will need to cover in more depth here to prepare you for the project: displaying pixels, taking keyboard input and system calls (`syscall`).

Displaying Pixels

The Frogger game will appear on the Bitmap Display in MARS (which you launch by selecting it in the MARS menu: Tools → Bitmap Display). This bitmap display is meant to simulate what would appear on a MIPS processor, given your assembly code.

The bitmap display is a 2D array of “units”, where each “unit” would be a pixel on the MIPS processor. As you can see from the image on the right, this window allows you to specify features of this display:


Bitmap Display

Unit Width in Pixels: 8
Unit Height in Pixels: 8
Display Width in Pixels: 256
Display Height in Pixels: 256
Base address for display: 0x10008000 ($gp)

- The unit width and height is like a zoom factor, indicating how many pixels on the bitmap display are used to represent a single pixel on the MIPS processor. In the image here, a single pixel on the MIPS processor would appear as an 8x8 box of a single colour on the display window here.
- The display width and height in pixels specified the width and height of the box in this bitmap display. The dimensions of computer screens can vary, so once you specify the dimensions you would like to use, your code will calculate the positions of the pixels to draw based on those dimensions.
- The "base address for display" indicates the location in memory that is used to display pixels on the screen, in a pattern called *row major order*:
    - If you write a colour value in memory at the base address, a pixel of that colour will appear in the top left corner of the Bitmap Display window.
    - Writing a colour value to the (base address + 4) will draw a pixel of that colour one pixel to the right of the first one.
    - Once you run out of pixel locations in the first row, the next pixel value will be written into the first column of the next row, and so on.
    - Code that illustrates this is provided in the Bitmap Starter Code section, later in this handout.

The “base address of display” needs to be set to memory location `0x10008000` each time you launch the bitmap display window. This is because the bitmap display window checks this location (and subsequent locations) in memory to know what pixels your code has asked it to display. If this drop-down box is left as its default value *(`static data`)*, the bitmap display will look for pixel values in the *"`.data`"* section of memory, which may cause the bitmap display to use your program instructions as colour data, leading to unexpected behaviour.

Bitmap Display Starter Code

To get you started, the code below provides a short demo, painting three units at different locations with different colours. Understand this demo and make it work in MARS.

```
# Demo for painting
#
# Bitmap Display Configuration:
# - Unit width in pixels: 8
# - Unit height in pixels: 8
# - Display width in pixels: 256
# - Display height in pixels: 256
# - Base Address for Display: 0x10008000 ($gp)
#
.data
      displayAddress:    .word 0x10008000
.text
      lw $t0, displayAddress   # $t0 stores the base address for display
      li $t1, 0xff0000   # $t1 stores the red colour code
      li $t2, 0x00ff00   # $t2 stores the green colour code
      li $t3, 0x0000ff   # $t3 stores the blue colour code

      sw $t1, 0($t0)      # paint the first (top-left) unit red.
      sw $t2, 4($t0)      # paint the second unit on the first row green. Why
$t0+4?
      sw $t3, 128($t0) # paint the first unit on the second row blue. Why +128?
Exit:
      li $v0, 10 # terminate the program gracefully
      syscall
```

Each pixel uses a 4-byte colour value, similar to the encoding used for pixels in Lab 7. In this case, the first byte isn’t used, but the next 8 bits store the red component, the 8 bits after that store the green component and the final 8 bits store the blue component (remember: 1 byte = 8 bits).

1. For example, `0x000000` is black, `0xff0000` is red and `0x00ff00` is green.
2. To paint a specific spot on the display with a specific colour, you need to:

a. Calculate the colour code you want using the combination of red, green and blue components,
b. Calculate the pixel location using the display width and height,
c. Finally, store that colour value in the correct memory address, using the `sw` instruction illustrated in the sample code below.

Tip: Pick a few key colour values (for water, frog, logs, etc) and consider storing them in specific registers or memory locations so that it's easy to put the colors into memory.

Fetching Keyboard Input

As illustrated in the bitmap display, MARS (and processors in general) uses **memory-mapped I/O (MMIO)**, meaning that certain locations in memory are used to communicate values between your assembly code and the underlying hardware. Interfacing with the keyboard is similar. When a key is pressed (called a *keystroke event*), the processor will tell you by setting a location in memory (address `0xffff0000`) to a value of `1`. This means that to check for a new keystroke event, you first need to check the contents of that memory location:

```
lw $t8, 0xffff0000
beq $t8, 1, keyboard_input
```

If that memory location has a value of 1, the ASCII value of the key that was pressed will be found in the next integer in memory. The code below is checking to see if the lowercase '`a`' was just pressed:

```
lw $t2, 0xffff0004
beq $t2, 0x61, respond_to_A
```

Syscall

The `syscall` instruction is needed to perform special built-in operations, namely invoking the random number generator and the sleep function (and exiting the program gracefully, as shown in the bitmap display sample code). The syscall instruction looks for a numerical code in register `$v0` and performs the operation corresponding to that code.

To invoke the **random number generator**, there are two services you can call:

- Service `41` produces a random integer (no range)
- Service `42` produces a random integer within a given range.

To do this, you put the value `41` or `42` into register `$v0`, then put the ID of the random number generator you want to use into `$a0` (since we're only using one random number generator, just use the value `0` here). If you selected service 42, you also have to enter the maximum value for this random integer into `$a1`.

Once the syscall instruction is complete, the pseudo-random number will be in `$a0`.

```
li $v0, 42
li $a0, 0
li $a1, 28
syscall
```

The other syscall service you will want to use is the **sleep operation**, which suspends the program for a given number of milliseconds. To invoke this service, the value `32` is placed in `$v0` and the number of milliseconds to wait is placed in `$a0`. The following code sample makes the processor wait for 1 second before proceeding to the next line:

```
li $v0, 32
li $a0, 1000
syscall
```

More details about these and other syscall functions can be found here.

Getting Started

This project must be completed individually, but you are encouraged to work with others when exploring approaches to your game. Keep in mind that you will be called upon to explain your implementation to your TAs when you demo your final game.

You will create an assembly program named `frogger.asm`. You'll be designing and implementing your code from scratch, starting with the starter code provided here.

Quick start: MARS

1. If you haven't downloaded it already, get MARS v4.5 here.
2. Open a new file called `frogger.asm` in MARS
3. Set up display: Tools > Bitmap display
    - Set parameters like unit width & height (8) and base address for display. Click "Connect to MIPS" once these are set.
4. Set up keyboard: Tools > Keyboard and Display MMIO Simulator
    - Click "Connect to MIPS"

...and then, to run your program:

5. Run > Assemble (see the memory addresses and values, check for bugs)
6. Run > Go (to start running your program)
7. Try entering characters (such as `w`, `a`, `s` or `d`) in Keyboard area (bottom white box) in Keyboard and Display MMIO Simulator window

Code Structure

Your code should store the location of the frog and all objects on the screen, ideally in memory (you want to reserve your registers for calculations and other operations). Make sure to determine what values you need to store and label the locations in memory where you'll be storing them (in the `.data` section)

Once your code starts, it should have a central processing loop that does the following:

1. **Check for keyboard input**
   a. Update the location of the frog accordingly
   b. Check for collision events (between the frog and the screen)
2. **Update the location of the logs, vehicles and other objects**
3. **Redraw the screen**
4. **Sleep.**
5. **Go back to Step #1**

How long a program sleeps depends on the program, but even the fastest games only update their display 60 times per second. Any faster and the human eye can't register the updates. So yes, even processors need their sleep.

Make sure to choose your display size and frame rate pragmatically. The simulated MIPS processor isn't super fast. If you have too many pixels on the display and too high a frame rate, the processor will have trouble keeping up with the computation. If you want to have a large display and fancy graphics in your game, you might consider optimizing your way of repainting the screen so that it does incremental updates instead of redrawing the whole screen; however, that may be quite a challenge.

General Tips

1. **Use memory for your variables.** The few registers aren't going to be enough for allocating all the different variables that you'll need for keeping track of the state of the game. Use the ".data" section (static data) of your code to declare as many variables as you need.

2. **Create reusable functions.** Instead of copy-and-pasting, write a function. Design the interface of your function (input arguments and return values) so that the function can be reused in a simple way.
3. **Create meaningful labels.** Meaningful labels for variables, functions and branch targets will make your code much easier to debug.
4. **Write comments.** Without proper comments, assembly programs tend to become incomprehensible quickly even for the authour of the program. It would be in your best interest to keep track of stack pointers and registers relevant to different components of your game.
5. **Start small.** Don't try to implement your whole game at once. Assembly programs are notoriously hard to debug, so add each feature one at a time and always save the previous working version before adding the next feature.
6. **Play the game**. Use the playable link from the first page to help you make decisions like when to move the logs and vehicles, how fast the frog should jump, etc.

Marking Scheme

This assignment has 5 milestones worth 20 marks total (4 points each), which are divided in the following way:

1. **Milestone 1: Draw the scene**
    a. Draw the background of the game, consisting of five regions:
        i. Starting region (at the bottom, where the frog starts off the game)
        ii. Road region (where vehicles travel back and forth)
        iii. Safe region (between road and water)
        iv. Water region (were logs and/or turtles travel back and forth)
        v. Goal region (at the top)
    b. Draw the objects of the game:
        i. The frog
        ii. The vehicles in the road (2 rows, with at least 2 objects in each row)
        iii. The floating logs and/or turtles in the water (same requirement as the road section)
    c. Repaint the screen ~60 times per second
2. **Milestone 2: Implement movement and controls**
    a. With keyboard input, make the frog move in all four directions
    b. Make the objects move in the road and the water
        i. The rows within the road and water move in opposite directions.
3. **Milestone 3: Collision detection**
    a. Frog resets to starting position if it collides with vehicles or lands in water
    b. If the frog reaches an empty goal region, mark that region as filled.

   c. End game once the player loses three frogs

4. **Milestone 4: Game features (one of the following combinations)**
   a. 4 easy features
   b. 2 easy features + 1 hard feature
   c. 2+ hard features
5. **Milestone 5: Additional features (one of the following combinations)**
   a. 7 easy features
   b. 5 easy features + 1 hard feature
   c. 3 easy features + 2 hard features
   d. 1 easy feature + 3 hard features
   e. 4+ hard features

Easy Features	**Hard Features**
Display the number of lives remaining.	Make a second level that starts after the player completes the first level.
After final player death, display game over/retry screen. Restart the game if the “retry” option is chosen.	Add sound effects for movement, collisions, game end and reaching the goal area.
Dynamic increase in difficulty (speed, obstacles, etc.) as game progresses	Have some of the floating objects sink and reappear (like arcade version)
Make objects (frog, logs, turtles, vehicles, etc) look more like the arcade version.	Add extra random hazards (alligators in the goal areas, spiders on the logs, etc)
Have objects in different rows move at different speeds.	Add powerups to scene (slowing down time, score booster, extra lives, etc)
Add a third row in each of the water and road sections.	Two-player mode (two sets of inputs controlling two frogs at the same time)
Display a death/respawn animation each time the player loses a frog.	Display the player’s score at the top of the screen.
Randomize the size and/or appearance of the logs and cars in the scene.	
Make the frog point in the direction that it’s traveling.	

We are open to features that we haven't thought of. If you would like to request a feature that is not on the list, please email the course coordinator. The list will be updated if requested features are approved.

Note: All requests regarding additional features must be sent before 10:00 PM on Dec 3, 2021. We will freeze the feature list and no longer accept new requests after this deadline.

Check-In Demo & Final Demo

The final demo for your project takes place on Tuesday, December 7th or Wednesday, December 8th (depending on your lab time). Each milestone you complete by that final demo earns you 20% of the project mark (4 marks each).

The second-last demo takes place on Tuesday, November 30th or Wednesday, December 1st, at which point you will demonstrate the basic game (Milestone 3). While we anticipate that some students might fall short of this goal, any student who doesn't have Milestone 1 completed by this deadline will incur a loss of 20% (4 out of 20) on your final mark for the project.

Required Preamble

The code you submit (`frogger.asm`) MUST include a preamble (with the format specified below) at the beginning of the file. The preamble includes information on the submitter, the configuration of the bitmap display, and the features that are implemented. This is necessary information for the TA to be able to mark your submission.

```
#####################################################################
#
# CSC258H5S Fall 2021 Assembly Final Project
# University of Toronto, St. George
#
# Student: Name, Student Number
#
# Bitmap Display Configuration:
# - Unit width in pixels: 8
# - Unit height in pixels: 8
# - Display width in pixels: 256
# - Display height in pixels: 256
# - Base Address for Display: 0x10008000 ($gp)
#
# Which milestone is reached in this submission?
# (See the assignment handout for descriptions of the milestones)
# - Milestone 1/2/3/4/5 (choose the one the applies)
```

```
#
# Which approved additional features have been implemented?
# (See the assignment handout for the list of additional features)
# 1. (fill in the feature, if any)
# 2. (fill in the feature, if any)
# 3. (fill in the feature, if any)
# ... (add more if necessary)
#
# Any additional information that the TA needs to know:
# - (write here, if any)
#
#####################################################################
```

Submission

You will submit your `frogger.asm` (only this one file) by using Quercus. You can submit the same filename multiple times and only the latest version will be marked. It is also a good idea to backup your code after completing each milestone or additional feature (e.g, use Git), to avoid the possibility that the completed work gets broken by later work. Again, make sure your code has the required preamble as specified above.

Late submissions are not allowed for this project, except for documented unusual circumstances.

Academic Integrity

Please note that ALL submissions will be checked for plagiarism. Make sure to maintain your academic integrity carefully, and protect your own work. It is much better to take the hit on a lower assignment mark (just submit something functional, even if incomplete), than risking much worse consequences by committing an academic offence.

Useful Resources

MIPS System Calls Table
MIPS Reference Card
MIPS API Reference
Assembly Slides (UTM)
ASCII Values

I still have no idea where to start! Help!

The secret to putting this game together is to start with something that works and add to that, one step at a time.

With that in mind, here are steps that you can take to build up the first few milestones:

1. **Start with the Bitmap Display Starter Code**. Copy and paste that code into your `frogger.asm` file. Make sure the bitmap display is visible and connected to MIPS, and set the correct values for height, width and base address. Run this, and make sure the pixels show up in the top left of the bitmap display.
2. **Get a piece of graph paper**. You'll need to plan how big your frog will be, so that you know how big everything else needs to be. Let every square on your graph paper represent a location that the frog can occupy in the game. Figure out where the safe spaces will be on the graph paper, and what the initial log and vehicle positions will be. Based on how many pixels high and wide your frog is, you can figure out how many pixels high and wide your overall game map will be. *You might need to change your bitmap display settings to fit your design*.
3. **Assuming that the top left of your map is pixel 0, mark significant parts of your map with pixel offset values**. When you're filling in a section of the bitmap display with a single colour, you'll need to iterate through the display memory from some starting location to an end location. It's good to have those marked on your graph paper so you don't have to keep calculating it in your head each time.
4. **Draw a rectangle.** To do this, take the Bitmap Display Starter Code you copied into your `frogger.asm` file and add code around the `sw` command to implement a loop. With each iteration, this loop code should store a single colour value at a different location in memory, and the code should loop as many times as there are pixels in the rectangle (this is why you need Step #3).
5. **Draw rectangles for the start region, the goal region and the safe section in the middle**. If you can put your rectangle code into a function, this would be the best approach here. If you're not ready for functions yet, then copy and paste the code from Step #4 until you learn about functions.
6. **Create a variable in memory that stores the frog location**. In the `.data` section of your code, create variables that store the X and Y location of your frog.
7. **After drawing your rectangles, draw your frog**. You need to use the X and Y values to calculate where the top left corner of your frog will be, and then colour the correct pixel locations to draw your frog.

8. **Create space in memory to store a row of vehicles.** How many pixels will a row of vehicles need? In your `.data` section, use the `.space` command to allocate enough space in memory for that many pixels and assign that space to a variable.

9. **Fill this allocated space with the pixels for the first vehicle row**. Right before the rectangle-drawing code, write code that fills in the space you just allocated in Step #8 with the pixel values for a single row of vehicles. Any spot that's occupied by a vehicle, set that spot to the pixel colour for your vehicle. Any spot that's empty, set it to the background colour for the road.

10. **Draw a vehicle row**. This is the same as drawing a rectangle, but the pixel values you use for the rectangle come from the allocated space you just filled in Step #9.

11. **Draw the other vehicle row, and the log rows**. Use Steps 8-10 to fill in the remaining vehicle and log rows.

12. **Move a row of vehicles or logs**. Use the sample sleep code to redraw the entire scene 60 times/second. Each time it redraws, change the pixels in your allocated memory space from Step 9 to shift all the pixels to the left or the right (wrapping any that fall off the edge to the other side).

13. **Move all the vehicle and log rows**.

14. **Move the frog**. Use the sample keyboard code to make the frog move forward one space in response to a single keypress.

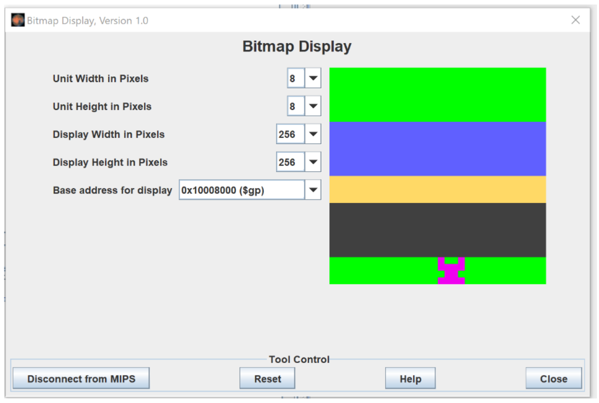
15. **Complete the other keyboard inputs**. Implement the other frog movements, as well as any other commands (pause, quit, etc).

16. **Collision detection, part 1**. If the frog's position overlaps with one of the vehicles on the road section, have the game restart.

17. **Collision detection, part 2**. If the frog's position doesn't overlap with the logs in the water section, have the game restart. Also, if a frog is on one of the logs, have the frog move with the log.

18. **Collision detection, part 3**. If the frog comes in contact with the goal section, trigger the win state.

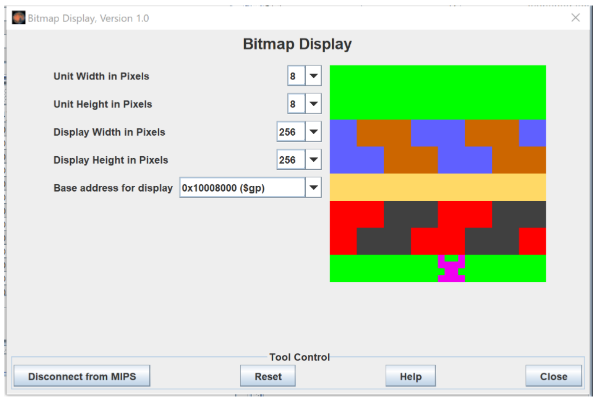
And most of all, make sure that you save the most recent working copy of your game before you work on adding something new! It's almost guaranteed that the changes you make to implement one of these steps will break things so badly that you'll need to restore a previous version. So be prepared, and good luck! 😁


Bitmap Display, Version 1.0

Bitmap Display

Unit Width in Pixels	8
Unit Height in Pixels	8
Display Width in Pixels	256
Display Height in Pixels	256
Base address for display	0x10008000 ($gp)

A bitmap display after Step 7.


Bitmap Display, Version 1.0

Bitmap Display

Unit Width in Pixels	8
Unit Height in Pixels	8
Display Width in Pixels	256
Display Height in Pixels	256
Base address for display	0x10008000 ($gp)

A bitmap display after Step 11.