Assembly Project – CSCB58 Winter 2021

Horizontal Scrolling Space / Shoot-em-up Game

Overview

In this project, you will build a horizontal scrolling space game using MIPS assembly. These games (also known as “shmups”) are a classic genre of games: you control a spaceship flying through space and avoiding obstacles, as well as optionally shooting, picking things up, and more. You have some level of freedom and we encourage you to be creative in how the game looks and feels. We set up the basic requirement, but it is your game! You can see several classic examples of such games for inspiration below (click links for videos). **You do not need to achieve this level of polish and sophistication!**

Gradius

R-Type

Astro Attack

Since we don't have access to physical computers with MIPS processors, you will develop and test your implementation in a simulated environment within MARS. We will use a simulated bitmap display and a simulated keyboard input.

The project has two milestones. In the **mid-way checkpoint**, you will demonstrate the basic features to your TA during your final interview slot at the last week of the term. For the **final version**, you will choose additional features to implement from a list of potential additions that change how the game looks and behaves. You will then submit the completed assembly program with a video link and TAs will later grade it. Note that projects are **individual**.

Important Info and Due Dates

- **Check Piazza post @235 frequently for FAQs and updates.**
- Checkpoint: Due during your usual TA interview slot on Week 12 (Apr 5 – Apr 9).
- Final submission: Due Monday, April 12, 2021, 11:59pm (on Quercus).

Version

- Mar 16, 2021: initial public version.

The Game

Basic Gameplay

The basic format of the game is simple: your spaceship is flying through space (perhaps towards the right side). There are multiple obstacles flying towards the ship that you need to avoid by moving the ship around the screen so as not to crash into the objects. As objects fly out of the screen, new ones appear. When you crash into obstacles, the ship suffers damage -- If you crash into too many obstacles, the game ends!

Game Controls

You use the keyboard to control the ship. While the **w** key is pressed, the ship will move up if it is not at the top edge of the screen. Similarly, the **a** key makes the ship move left, the **d** key moves right, and **s** moves down (unless the ship is already at that edge of the screen). See "Technical Background" below to see how to read the status of keys.

Basic Demo

Click here for a video demonstration of a basic version of what you are supposed to implement (it also shows how to set up the MARS simulator for the project). The demo only shows the features we expect for the Midway Checkpoint version (explained below); for the Final Submission, you will implement additional features. Note it is OK if your game looks and behaves somewhat differently from our demo, as long as the features are there!

Additional Features

The above only describes the basic gameplay. To get full marks, you will need to implement additional features from a list. There are many possible additions to the gameplay, and here are a few examples:

- Different flying patterns and directions for obstacles.
- Adding "enemy ships" that move around in a pattern.
- Adding the ability to shoot obstacles / enemy ships.
- Picking up powerups that "repair" your ship, change obstacles size and speed, and so on.

See the list of additional features at the end.

Creativity and Limits

While you are not required to do so, we encourage you to be creative within the framework of the game and features we have required. Many of the features are defined generally on purpose, to allow you some freedom. Moreover, you have a lot of control in how the game looks and feels. You don't have to emulate our demo!

As you consider your creativity, be aware of the limits of what we are working with. The MARS simulator is not very fast. This implies there are only so many MIPS instructions you can execute in one second, which limits the complexity of your game and graphics. You may be able to come up with interesting tricks to make things run faster, but we are not marking you based on this!

Technical Background

You will create this game using the MIPS assembly language taught in class and the MARS. However, there are a few 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 Using a Framebuffer

To display things on the screen, we will use a *framebuffer*: an area in memory shown as a 2D array of "units", where each "unit" is a small box of pixels of a single colour. Figure 1 on the right shows a framebuffer of width 10 and height 8 units.

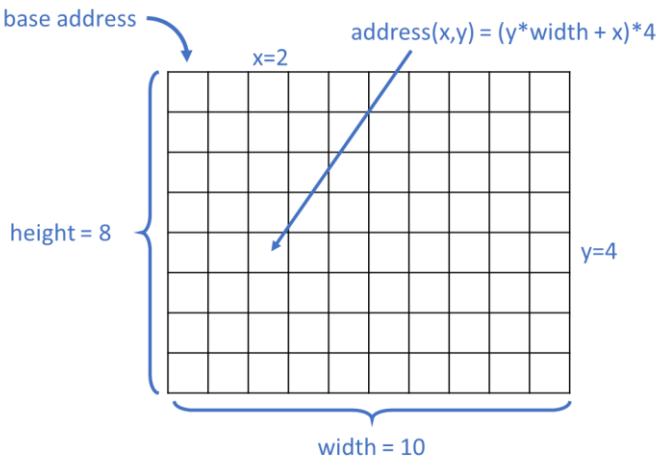

Figure 1: Framebuffer

Units are stored as an array in memory: every 4-byte word in that array is mapped to a single on-screen "unit". This is known as a *framebuffer*, because the buffer (the region of memory) controls what is shown on the display. The address of this frame buffer array is the *base address* of the display. The unit at the top-left corner of the bitmap is located at the base address, followed by the rest of the top row in the subsequent locations in memory. This is followed by the units of the second row, the third row and so on (referred to as row major order).

To set the colour of a single unit, you will write a 4-byte colour value to the corresponding location in memory. Each 4-byte value has the following structure: **0x00RRGGBB** , where 00 are just zeros, RR is an 8-bit colour value for the red component, GG are the 8-bits for the green components, and BB are the 8-bits for the blue component. For example, `0x00000000` is black, `0x00ff0000` is bright red, `0x0000ff00` is green, and `0x00ffff00` is yellow. To paint a specific spot on the display with a specific colour, you need to calculate the correct colour code and store it at the right memory address (perhaps using the `sw` instruction).

The MARS Bitmap Display

MARS allows us to map a framebuffer in memory to pixels on the screen using the **Bitmap Display** tool, which you launch by selecting it in the MARS menu: **Tools → Bitmap Display**. The Bitmap Display window is shown in **Error! Reference source not found.**.

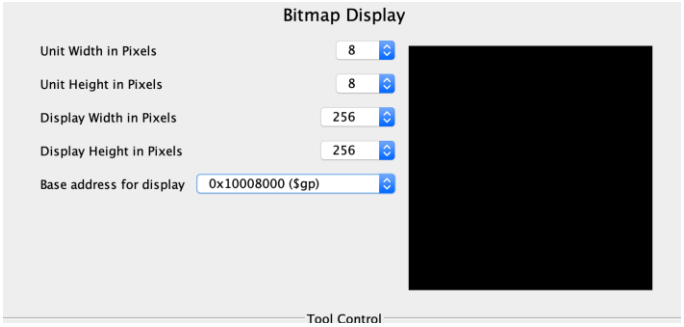

Figure 2: The MARS Bitmap Display

- Specify the actual on-screen width and height of each unit. The screenshot is configured to show each framebuffer unit as 8x8 block on your screen.
- You need to set the dimensions of the overall display: in the example above the framebuffer is of size 32x32 units, since we configured the bitmap display to have width and height of 256 pixels, and units of 8x8 pixels. Make sure the sizes match your assembly code.
- You need to tell MARS the base address for the framebuffer in hexadecimal. In screenshot above, this is memory location `0x10008000` in the screenshot. This means that the unit in the top-left corner is at address `0x10008000`, the first unit in the second row is at address `0x10008080` and the unit in the bottom-right corner is at address `0x10008ffc`.
- Remember to click "Connect to MIPS" so that the tool connects to the simulated CPU.

Tip: We recommend using `0x10008000` as the base address for your framebuffer. When using the Bitmap Display, make sure to change the "base location of display" field. If you set it to the default value *(`static data`)* provided by the Bitmap Display dialog, this will refer to the "*`.data`*" section of memory and may cause the display data you write to overlap with instructions that define your program, leading to unexpected bugs.

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.

The assembly directive `.eqv` defines a numeric constant which you can use instead of writing the number manually (similar to `#define` in C, but much more primitive) You can use it to define common useful constants such as fixed addresses, colours, number of obstacles, etc.

```
# Bitmap display starter code
#
# 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)
#
.eqv  BASE_ADDRESS            0x10008000

.text
      li $t0, BASE_ADDRESS    # $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?

      li $v0, 10 # terminate the program gracefully
```

```
    syscall
```

Keyboard

This project will use the MARS **Keyboard and MMIO Simulator** to take in these keyboard inputs (**Tools → Keyboard and MMIO Simulator**). To use it when playing, make sure to click inside the lower window titled "KEYBOARD". As with the bitmap display, remember to "Connect to MIPS".

Note we cannot use system calls to read the keyboard input, because syscalls will block your program from executing (they are also slower).

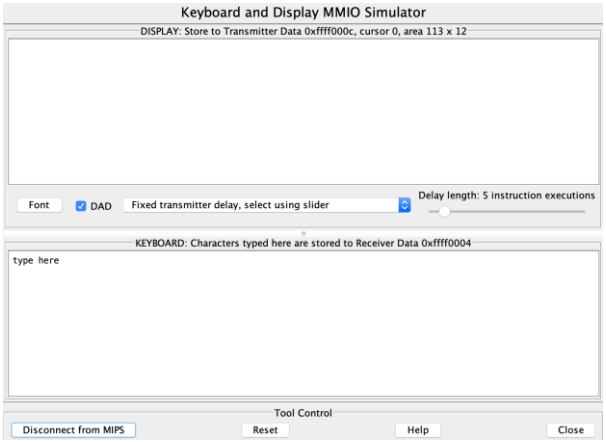

Figure 3: MARS Keyboard and MMIO Simulator

Fetching Keyboard Input

MARS uses *memory-mapped I/O* (MMIO) for keyboard. If a key has been pressed (called a *keystroke event*), the processor will tell you by setting a word in memory (at address `0xffff0000`) to a value of `1`. To check for a new key press, you first need to check the contents of that memory location:

```
li $t9, 0xffff0000
lw $t8, 0($t9)
beq $t8, 1, keypress_happened
```

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

```
lw $t2, 4($t9) # this assumes $t9 is set to 0xfff0000 from before
beq $t2, 0x61, respond_to_a   # ASCII code of 'a' is 0x61 or 97 in decimal
```

Useful Syscalls

In addition to writing the bitmap display through memory, the `syscall` instruction will be needed to perform special built-in operations, namely invoking the random number generator and the sleep function.

To invoke the **random number generator**, you can use service 41 to produce a random integer with no range limit, or service 42 to produce a random integer within a given range.

To do this, 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`:

```
li $v0, 32
li $a0, 1000   # Wait one second (1000 milliseconds)
syscall
```

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

Your Code

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 `game.asm`. You'll design your program from scratch, but you must begin your file with the preamble starter code we include below.

1. Open a new file in MARS, call it `game.asm`.
2. Set up display: Tools > Bitmap display (the demo shows how to do this)
    - Set parameters like unit width and height (we recommend 8) and base address for display.
    - Click "Connect to MIPS" once these are set.
3. Set up keyboard: Tools > Keyboard and Display MMIO Simulator (see demo)
    - Click "Connect to MIPS"

...and then, to run and test your program:

4. Run > Assemble (see the memory addresses and values, check for bugs)
5. Run > Go (to start the run)
6. Input the character `a` or `d` or `w` or `s` in Keyboard area (bottom white box) in Keyboard and Display MMIO Simulator window.

Code Structure

Your code should store the location of the ship and other entities (and any other information you need) in memory or registers. We particularly recommend arrays for storing things like obstacles. 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)

At the beginning of your program, **clear the screen** and **initialize the state of the game**. You will then switch to a central processing loop of the game (the "**main loop**"). Everyone's main loop looks a little

different, but in general your loop will eventually need to do most if not all of the following operations (**not necessarily in this order!**):

- Check for keyboard input and update ship location.
- Update obstacle location.
- Check for various collisions (e.g., between ship and obstacles).
- Update other game state and end of game.
- Erase objects from the old position on the screen.
- Redraw objects in the new position on the screen.

At the end of each iteration, your main loop should sleep for a short time and go back to step 1.

Sleep and Display Refresh Rate

For animations, we generally need to update the display between 20 to 60 times per second (we recommend sleeping for 40ms, at least initially, which is a 25Hz update rate). When developing your game, you may find it occasionally useful to set it to a very high number to help debugging. We recommend to use a constant (`.eqv`) to make it easy to change the wait time.

Make sure to choose your display size and frame rate pragmatically. The simulated MIPS processor isn't very 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! Moreover, sleep time can affect the speed and difficulty of your game.

Tip: Your display may flicker (blink on and off), because it takes some time between erasing an object and redrawing it at the new position. There are all kinds of ways to prevent this, but many of them require special hardware. **We are not marking you based on the smoothness of the display as long as it's reasonably playable, so don't worry about it.** See Marking below.

General Tips for Success

1. **Choose storage wisely**. Most of your variable will be stored in memory (`.data`), because you only have a few registers, and they are not 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. In particular, arrays are very useful. Nevertheless, it might make sense to devote a few registers to very common variables or values that you need! Calling conventions can help you there.
2. **Create reusable functions for code that you reuse frequently.** Instead of copy-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. **…but pay attention to performance.** Calling functions comes at a cost. Use functions when you need to (using the same code many times), but not when you don't. Think about calling conventions that make sense for you and are efficient. For example, you may be able to avoid using the stack entirely!
4. **Flicker and clearing the screen**. Erasing the entire screen every frame is a slow process and will cause lots of flickering. It is better to just erase the part you need to erase. The more time that passes between erase and redraw, the more flickering you will have.
5. **Create meaningful labels.** Meaningful labels for variables, functions and branch targets will make your code much easier to debug.
6. **Write comments.** Without proper comments, assembly programs tend to become incomprehensible spaghetti alarmingly quickly, even for the author of the program. It would be in your best interest to keep track of registers, variables, and stack pointers (if you use it) relevant to different components of your game.
7. **Start small and build upwards.** 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 test them. Always save the previous working version before adding the next feature. Use source control (Git, SVN, etc.).
8. **Debug.** Debug your code cleverly. For example, you may find it useful to change the sleep time between frames to 1000ms so you can see things advance slowly, or even to wait for a key. It makes sense to sometimes add code just for debugging, and then remove it later.
9. **Play your game**. Take some time to make conscious decisions about ship movement speed, obstacle movement, difficulty, sleep time, etc.
10. **Have fun with programming.** Try to do new things and make the game yours! Writing a game purely in assembly language is an accomplishment you should be proud of.

Required Preamble

The code you submit (`game.asm`) ***must*** include at the beginning of the file the lines shown below, in the same format. This preamble includes information on the submitter, the configuration of the bitmap display, and the features that are implemented, and a link to your video demonstration (more on this later). **This is necessary information for the TA to be able to mark your submission.**

```
################################################################
#
# CSCB58 Winter 2021 Assembly Final Project
# University of Toronto, Scarborough
#
# Student: Name, Student Number, UTorID
#
# Bitmap Display Configuration:
# - Unit width in pixels: 8 (update this as needed)
# - Unit height in pixels: 8 (update this as needed)
# - Display width in pixels: 256 (update this as needed)
```

```
# - Display height in pixels: 256 (update this as needed)
# - Base Address for Display: 0x10008000 ($gp)
#
# Which milestones have been reached in this submission?
# (See the assignment handout for descriptions of the milestones)
# - Milestone 1/2/3/4 (choose the one the applies)
#
# Which approved features have been implemented for milestone 4?
# (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)
#
# Link to video demonstration for final submission:
# - (insert YouTube / MyMedia / other URL here). Make sure we can view it!
#
# Are you OK with us sharing the video with people outside course staff?
# - yes / no / yes, and please share this project github link as well!
#
# Any additional information that the TA needs to know:
# - (write here, if any)
#
##################################################################
```

Project Submission and Marking

Projects are individual. This assignment is worth 15 points. 8 of this will be evaluated during the check-in demo on the last week of classes depending on your progress, and the other 7 will be for the functionality of your final submission. To better give you a sense of progress, the assignment is broken down into 4 milestones as follows. For milestones 1-3 you need to implement **all** features; for milestone 4 (only) you choose a set of features to implement.

1. **Milestone 1: basic animations [4 marks]**
    a. Show the obstacles moving across the screen and the ship. This means continually redrawing the screen with the appropriate assets. Your game's display resolution (framebuffer size) should be **at least 32 units high and 32 units wide**.
    b. There should be the spaceship and at least 3 obstacles. Both the ship and at least 3 obstacles need to be **at least 3 framebuffer units big** (more is allowed).
    c. When obstacles exit the screen, make sure new ones appear. Use a random number generator to make it random.
2. **Milestone 2: movement controls and collision detection [4 marks]**
    a. Player can move the ship around the screen using the movement keys. Make sure the ship cannot move past the edges of the screen!

b. Collision detection: detect when an obstacle is touching the ship. There are several ways to do this, and you don't have to be 100% accurate as long as they are very close.
c. Show when a collision has happened (anything is fine as long as it's obvious to the player).

⇨ In your midway checkpoint session, you need to demonstrate to your TA a working implementation of the first two milestones; see Basic Demo above for what that might look like. In addition, you will need to have at least started on the other milestones.

3. **Milestone 3: Finished game and user interface. [4 marks]**
   a. The game should keep track of collisions with obstacles.
   b. Show the remaining "health" of the ship on the screen during the game.
   c. If too many collisions have happened, show an appropriate "game over" screen.
   d. Allow restarting the game at any point by pressing the p key on the keyboard.
4. **Milestone 4: Game features and polish [3 marks].**
   Implement at least **3 features** from the list below. Your preamble should list specifically what features you implemented.
   a. Different levels: after some time, the player moves to the next "level", which will include a different set of obstacles types, sizes, and looks.
   b. Increase in difficulty as game progresses. Difficulty can be achieved by making things faster, adding more obstacles, making obstacles larger, etc.
   c. Scoring system: add a score to the game based on survival time, near misses, or any idea you may have. You must show the final score on the game-over screen for this to count!
   d. Add "pick-ups" that the ship can pick up (at least 2 types). Examples: temporary shield, bonus points, make everything slower, repair ship, and more.
   e. Enemy ships – some obstacles look different and move in "unnatural", difficult, or surprising patterns.
   f. Shoot obstacles/enemy ships: add ability to shoot obstacles using the keyboard.
   g. Smooth graphics: prevent flicker by carefully erasing and/or redrawing only the parts to the frame buffer that have changed. This requires some effort.

⇨ If you would like to request a feature not on the list, post on Piazza by end of Apr 1st. Note that not all requests will be approved!

Midway Checkpoint

In the last week of classes (Apr 5 – Apr 9), there will be a check-in session with your TA in your usual designated slot. By this point, you are expected to have **completed** the first two milestones at least, and have started working on some of the others. You will need to demonstrate your running code, and we will briefly ask you about how you have chosen to design certain parts of your game, and what your plan is for completing it. You do not need to submit the code for the midway checkpoint.

Final Submission

The deadline for final submission is going to be on the last day of classes, April 12 at 11:59pm. As such, we won't have an opportunity to meet to discuss your project. To help us evaluate your final project (so we don't miss anything), you need to submit a short video walking us through your project. Any final submission that does not include a video will ***not be marked***. If you have concerns with this, please get in touch with us *immediately.*

You will submit your `game.asm` (only this one file) to Quercus. You will host the videos externally and add a link to it in your submission. Look below for additional details on what to include in the file and the video. You can submit the same filename multiple times and only the latest version before the deadline 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.

As detailed in the term work policy, late submissions are not allowed for this project, except in for documented and unusual circumstances.

Required Video Demonstration

You will need to include a short video (around 5 to 7 minutes, no more than 10) walking us through your project. This is to help us properly evaluate you and not miss any features you might have implemented. In the video, you should:

1. Demonstrate that the basic functionality works (movement, jumping, ending the game, ...) and all the features in milestones 3.
2. Demonstrate the functionality of the additional features implemented for milestone #4 (remember to also list them in the preamble).
3. If you were unable to complete all milestones, explain what difficulties you encountered and show what progress you had on those features (so we can partially assign marks)
4. Tell us any other information you think you would be useful to us while we are evaluating your work.

Use screen-recording software for the demo instead of taking a video of your screen with your phone, if possible.

For sharing the video with us you will send use a URL (in the preamble). UofT provides a media sharing service you can use to host the video, or you can use YouTube and similar services of your choice. Another option is Office 365, provided by the university, also includes OneDrive which allows sharing videos as well. It's up to you! Regardless of what service you use, make sure to include the URL in the preamble of the submitted file, and **make sure the video is viewable by the course staff**.

Publicly sharing videos: do let us know in your preamble if you are OK with us sharing the video link with people outside course staff (e.g., future students). **It's up too you and won't affect your grade!**

Marking

Marking is based on completing features and answering oral questions. Your game does not need to be very smooth, polished, or even fun, to get full marks. As long as the game is playable and working, that's fine! However, your game needs to playable, which means (for example) we need to be able to see where things are, animations cannot be so slow or so flickering that we cannot see where things are, and the game cannot be so hard that it ends in 5 seconds. **Again, we are going to be very lenient on this, but your game does need to be playable.**

Bonus for Polish

We may decide to give an additional small bonus to students whose projects exceed our expectations. **You do not need this bonus to get full marks!** This bonus will be decided on a case-by-case basis and will depend on how well you have executed features you have implemented and how polished your overall game is beyond the minimum. For example, you may have made tangible, visible improvement to the graphics using bigger resolution, animations, and nice UI.

Academic Integrity

It is fine and even encouraged to discuss ideas, but sharing code is forbidden. All the code you submit must be your own.

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.

Remember that **sharing your code is also a violation**, not just using someone else's. If you are using web-based version control such as Github, ensure your repositories are private, and do not otherwise let anyone else see your code.

See the policy on the course info sheet for more details.

Useful Resources

- Assembly slides on Quercus
- Quercus assembly resource page
- MIPS System Calls Table