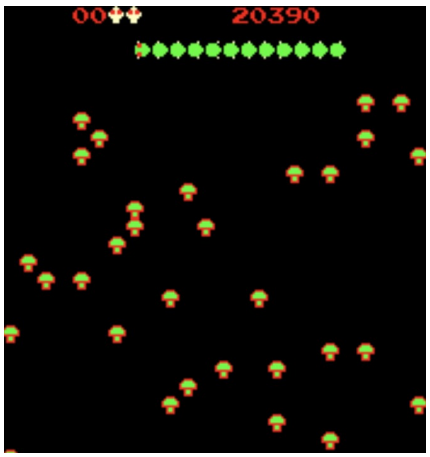
CSC258 Assembly Final Project: Centipede

Due dates:

- Check-in Demo: Mon/Wed, March 29/31
- Final Demo: Mon/Wed, April 5/7
  All demos and submissions must be completed individually.

Update 2021-03-26: Please refer to updated Marking Scheme (in green)

A) Overview



In this project, we will implement a modified version of the popular 1980 Atari game Centipede using MIPS assembly. Familiarize yourself with the game here (https://www.youtube.com/watch?v=V7XEmf02zEM) or here (Flash required; https://my.ign.com/atari/centipede).

Since we don't have access to physical computers with MIPS processors, we will test our implementation in a simulated environment within MARS, i.e., a simulated bitmap display and a simulated keyboard input.

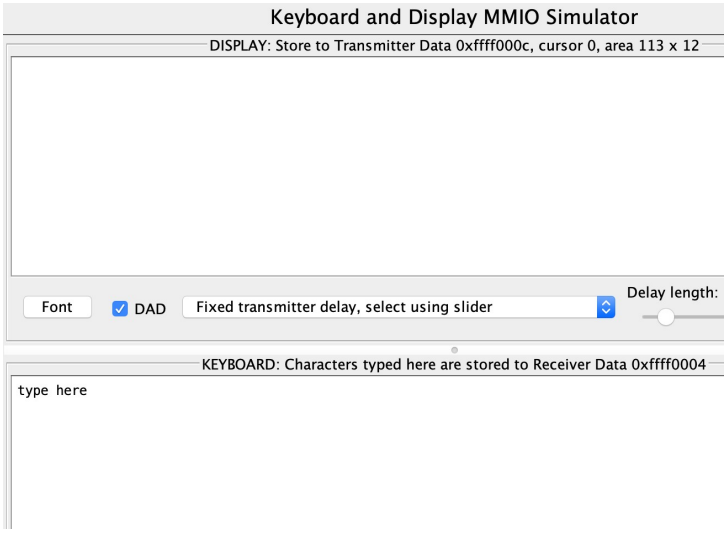
B) How to Play Centipede

The goal of this game is to get the highest number of points before the Bug Blaster loses all its lives; the Centipede head is worth 100, each body is worth 10, and each Mushroom 300. The Centipede containing 10 segments initially wriggles from the top to the bottom and left to right of the screen, descending one level every time a Mushroom interferes with its trajectory. Mushrooms that determine the direction and place on the screen of the Centipede appear when a part of the Centipede is blasted by a Dart and disappear after being shot by the Bug Blaster. Each segment blasted off the Centipede turns into a Mushroom at that location. Blasting in between Centipede segments separates the Centipede into two parts that continue wriggling in the current direction, respectively. When all Centipede segments are blasted and at the bottom, a new Centipede appears starting from the top downwards. The game ends if the Bug Blaster hits a Flea (drops vertically from the top of the screen to the bottom) or loses all its lives (total of 3).

C) Game Controls

There are 3 buttons used to control the Bug Blaster:

- The "j" key makes the Bug Blaster move to the left,
- The "k" key makes the Bug Blaster move to the right.
- The "x" key makes the Bug Blaster shoot at the centipede.



This project will use the Keyboard and MMIO Simulator to take in these keyboard inputs. In addition to the keys listed above, the "s" key will be used to start and restart the game as needed.

When no key is pressed, the Bug Blaster is stationary at the bottom of the screen. If it hits a Flea at any point, the game ends.

D) Project Goals

This handout describes the basic contents and behaviours of the game. If you implement something similar, you'll get part marks for this project. To get full credit, you will need to implement additional features to bring it closer to the actual game. More details on this below.

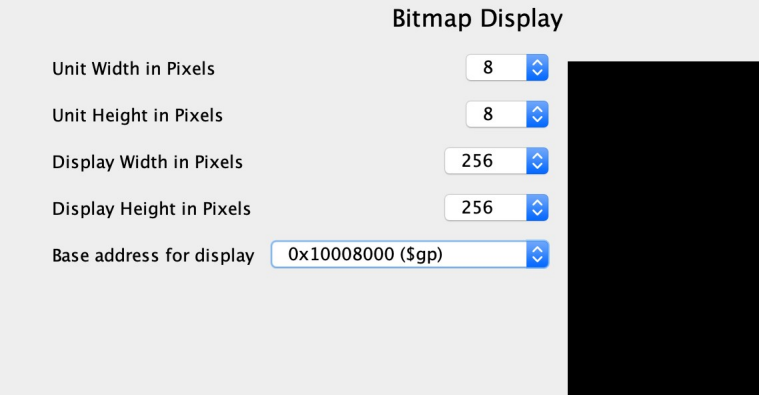
E) Technical Background

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`).

E.1) Displaying Pixels

The Centipede game will appear on the Bitmap Display in MARS (which you launch by selecting it in the MARS menu: Tools → Bitmap Display)

The bitmap display is a 2D array of "units", where each "unit" is a small box of pixels of a single colour. These units are stored in memory, starting at an address called the "base address of display". The Bitmap Display window allows you to specify the width and height of these units, the dimension of the overall display and the base address for the image in hexadecimal (see the Bitmap Display window on the right).



The 2D array of units for this bitmap display are stored in a section of memory, starting at the "base address of display" (memory location `0x10008000` in the example above). To colour a single unit in the bitmap, you must write a 4-byte colour value into the corresponding location in memory.

- The unit at the top-left corner of the bitmap is located at the base address in memory, 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*).
- The size of the array in memory is equal to the total number of units in the display, multiplied by 4 (each colour value is 4 bytes long).
    - For example, in the configuration in the above image, each unit is 8 pixels x 8 pixels and there are 32x32 = 1024 units on the display (since 256/8 = 32).

    - 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`.
- Each 4-byte value stored in memory represents a unit's colour code, similar to the encoding used for pixels in Lab 7. In this case, the first 8 bits aren'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.
    - For example, `0x000000` is black `0xff0000` is red and `0x00ff00` is green. 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).

When setting up the Bitmap Display dialog above, you must 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.

```
# Demo for painting
#
# Bitmap Display Configuration:
# - Unit width in pixels: 8
# - Unit height in pixels: 8
# - Display width 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
```

Tip: Pick a few key colour values (for background, Bug Blaster, Centipede, Mushroom) and consider storing them in specific registers or memory locations so that it's easy to put the colors into memory.

E.2) Fetching Keyboard Input

MARS (and processors in general) uses **memory-mapped I/O (MMIO)**. If a key has been pressed (called a *keystroke event*), the processor will tell you by setting a location in memory (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:

```
1 lw $t8, 0xffff0000
2 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:

```
1 lw $t2, 0xffff0004
2 beq $t2, 0x61, respond_to_a
```

E.3) Syscall

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**, 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`.

```
1 li $v0, 42
2 li $a0, 0
3 li $a1, 28
4 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`:

```
1 li $v0, 32
2 li $a0, 1000
3 syscall
```

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

F) 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 `centipede.s`. There is no starter code; you'll design your program from scratch.

F.1) Quick start: MARS

1. If you haven't downloaded it already, get MARS v4.5 here.
2. Open a new file called `centipede.s` 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 the run)
7. Input the character `j` or `k` or `x` in Keyboard area (bottom white box) in Keyboard and Display MMIO Simulator window

G) Code Structure

Your code should store the location of the Centipede, Bug Blaster, Fleas, and all Mushrooms on the screen, ideally in memory (i.e. 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 Bug Blaster accordingly
    b. Check for collision events (between the Bug Blaster and Fleas, blasted Darts and Mushrooms / Centipede segments, Mushrooms and Centipede)
2. **Update the location of all Centipede parts and other moving 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.

H) 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-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 author 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.

I) Marking Scheme

This project has 3 milestones worth 20 marks in total ~~5 milestones worth 20 marks total (4 points each)~~, which are divided in the following way:

1. **Milestone 1: Create animations (7 marks out of 20)**
    a. Continually repaint the screen with the appropriate assets
    b. Draw new location of Centipede (10 segments with distinctive head segment, zigzag movement), Bug Blaster, and Mushrooms
    c. If Centipede is at the bottom of the screen, it continues wriggling and invading the Bug Blaster's personal space
2. **Milestone 2: Implement movement controls (6 marks out of 20)**
    a. With keyboard input, move the Bug Blaster along the bottom of the screen
    b. Shoot out Darts when `x` is pressed
3. **Milestone 3: Basic running version (7 marks out of 20)**
    a. Random Flea movement and initial Mushroom location generator
    b. Centipede dies after being hit by dart 3 times. ~~Centipede segments turn into mushrooms if shot and "break" if shot in between head and tail.~~ You do not have to change your implementation if you have already done this according to the original specification~~.~~
    c. Terminate program if Flea intersects a Bug Blaster
    d. Game over / retry option

BONUS MILESTONES (worth a maximum of extra 3% total on top of total project marks)

4. **Milestone 4: Game features (at least 2)**
    a. Centipede segments turn into mushrooms if shot and "break" if shot in between head and tail
    b. Scoreboard / score count
        i. Display on the screen, increment when Dart blasts something
    c. Count number of lives (default 5) and display with icons
        i. Terminate game if lives exhausted

d. Different levels
    i. Add more Mushrooms, more Fleas, longer Centipede length with individually moving segments
e. Dynamic increase in difficulty (speed, obstacles, shapes etc.) as game progresses
    i. Must blast Mushroom 4 times to count as ridding it

5. **Milestone 5: Additional features (at least 3)**
    a. Realistic physics:
        i. Speed up / slow down Centipede wriggling rate or collision response according to some metric (e.g. gravity)
    b. More obstacle types:
        i. Spiders (zigzags the screen eating Mushrooms)
        ii. Scorpions (zigzags turning every mushroom into a poisonous one, if Centipede collides, all Centipede segments descend vertically)
            1. Only need to shoot Centipede head to revert back to original horizontal movement
        iii. Other types, distinguished by different colours
    c. Alternate scoring for blasted elements:
        i. Mushrooms are worth 300, 600, or 900 points based on how closely they were shot by the Darts
        ii. Scorpions shot are worth 1000 points
            1. When Bug Blaster is destroyed by an obstacle, all poisonous Mushrooms are reverted
            2. Each reverted mushroom is 5 points
        iii. Every 12000 points == an extra life
    d. Fancier graphics:
        i. Make it nicer than the demo
    e. Dynamic on-screen notifications:
        i. "Awesome!", "Poggers!", "Wow!"
    f. Player names
        i. Allow player to input this prior to starting the game
    g. Two Bug Blasters
        i. Separate keyboard inputs to control different entities

Check-In Demo & Final Demo

The entire project will be evaluated during your demo time on Mon/Wed, April 5th/7th. Before that though, there is a check-in demo on Mon/Wed, 29th/31st, at which point you MUST have

Milestone #1 completed. Failure to meet 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 (`centipede.s`) 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.

```
#####################################################################
#
# CSC258H Winter 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 project 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 project 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 `centipede.s` (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 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