# CSCB58 Assembly Final Project:  Doodle Jump

## Due dates:

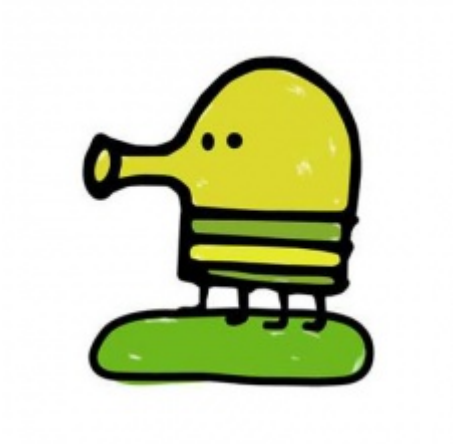
- Final submission:  Monday, Dec 7, 2020, 11:59pm.
- Check-in Demo:  Practical session, Week 12 (Nov 30 - Dec 4)
  All submissions must be completed individually.

## Document Updates

- **Nov 13, 2020**: Uploaded original project handout.
- **Nov 18 2020**: Fixed Milestone 5a, added Milestone 5d(iii)
- **Nov 20 2020**: Updated demo video to a better one
- **Dec 02 2020**: Added blurb on possible bonus 10%

## Overview

In this project, we will implement the popular mobile game Doodle Jump using MIPS assembly. You may familiarize yourself with the game 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 MARS, i.e., a simulated bitmap display and a simulated keyboard input.

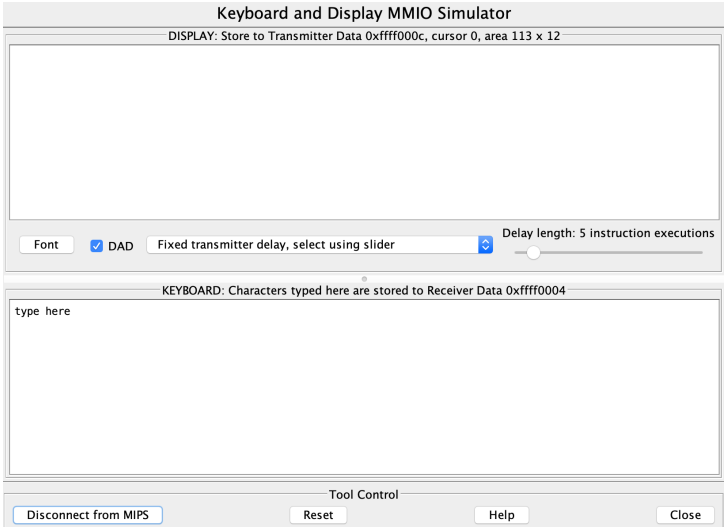
## How to Play Doodle Jump

The goal of this game is to see how high the Doodler can get by jumping up from platform to platform. The camera follows the Doodler as it jumps to higher platforms, with new platforms appearing at the top of the screen. The game ends if the Doodler misses the platforms and lands on the bottom edge of the screen.

## Game Controls

There are two buttons used to control the Doodler while in midair:

- The "`j`" key makes the Doodler move to the left,
- The "`k`" key makes the Doodler 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, the "`s`" key will be used to start and restart the game as needed.

When no key is pressed, the Doodler falls straight down until it hits a platform or the bottom of the screen. If it hits a platform, the Doodler bounces up high into the air, but if the Doodler hits the bottom of the screen, the game ends.

## 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.

## (Basic) Demo

Below is a video demonstration of a really basic version of what you're supposed to implement. Make this nicer with more features implemented in the later milestones!

MyMedia

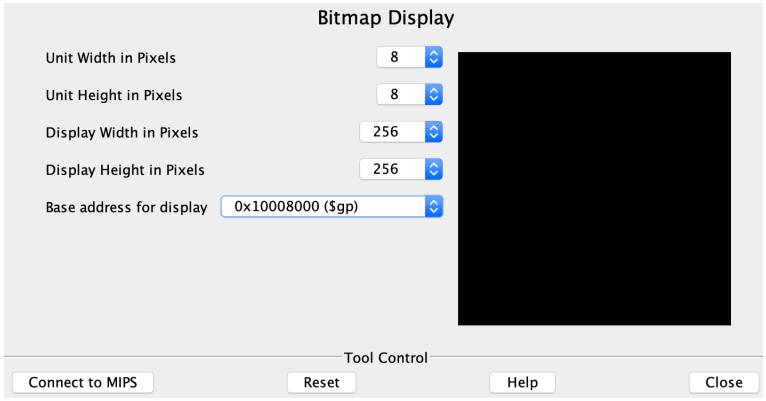
# Technical Background

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 Doodle Jump 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
# - 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
```

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

## *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:

```
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
```

## *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**, 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.

## 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 `doodlejump.s`. There is no starter code; you'll design your program from scratch.

## Quick start: MARS

1. If you haven't downloaded it already, get MARS v4.5 here.
2. Open a new file called `doodlejump.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` in Keyboard area (bottom white box) in Keyboard and Display MMIO Simulator window

## Code Structure

Your code should store the location of the Doodler and all platforms 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 Doodler accordingly
    b. Check for collision events (between the Doodler and the screen)

2. **Update the location of all platforms 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-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.
6. **Play the game**. Use the playable link from the first page to help you make decisions like when to move the platforms, how fast the Doodler should jump, etc.

## Marking Scheme

This assignment is worth 15 points. 50% of this will be evaluated during the check-in demo on the last week of classes depending on your progress, and the other 50% will be for the functionality of your final submission. To better give you a sense of progress, the assignment is broken down into 5 milestones (~3 marks each) divided as follows:

1. **Milestone 1: Create animations**
    a. Continually repaint the screen with the appropriate assets
    b. Draw new location of platforms (3 minimum) and Doodler properly

2. **Milestone 2: Implement movement controls**
    a. Doodler should be able to “jump” off a platform
    b. Change animation direction based on input keys

3. **Milestone 3: Basic running version (similar to demo)**
    a. Random platform generator / scrolling the screen
    b. End game and wait for restart if Doodle jumps onto illegal area

4. **Milestone 4: Game features (at least 2)**
    a. Scoreboard / score count
        i. Display on the screen
    b. Different levels
        i. Add more platforms, obstacles types etc.
    c. Dynamic increase in difficulty (speed, obstacles, shapes etc.) as game progresses

5. **Milestone 5: Additional features (at least 3)**
    a. Realistic physics:
        i. Speed up / slow down jump rate according to some metric
    b. More platform types:
        i. Moving blocks
        ii. “Fragile" blocks (broken when jumped upon)
        iii. Other types, distinguished by different colours
    c. Boosting / power-ups:
        i. Rocket suit
        ii. Springs
    d. Fancier graphics:
        i. Make it nicer than the demo
        ii. Add start / game over / pause screens with cool graphics.
    e. Dynamic on-screen notifications:
        i. "Awesome!", "Poggers!", "Wow!"
    f. Two Doodles
        i. Separate keyboard inputs to control different entities
    g. Opponents / lethal creatures
    h. Shields (for protection of Doodler)
    i. Shooting (of Doodler)

If you would like to request a feature that is not on the list, please email the assignment coordinator (*mustafa@cs.toronto.edu)*. The list will be updated if requested features are approved.

*(Updated Dec 2)* In addition, we have decided to give an additional 10% bonus to any students whose projects exceed all our expectations. This 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.

*Note: All requests regarding additional features must be sent before 10:00 PM on Dec 1, 2020. We will freeze the feature list and no longer accept new requests after this deadline.*

### Check-In Demo & Final Demo

In the last week of classes (Nov 30 - Dec 4), there will be a check-in session with your TA in the designated practical time. By this point, you are expected to have at least completed the first 2 milestones, and have started working on some of the others. 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.

The deadline for final submission is going to be on the last day of classes, December 7th at 11:59pm. As such, we won't have an opportunity to meet to discuss your project. In order to help us evaluate your final project (so we don't miss anything), you will be expected to submit a short (5-10) minute video walking us through your project. Any project submission that does not include a video will ***not be marked***. If you have concerns with this, please get in touch with us *now.*

## Submission

You will submit your `doodlejump.s` (only this one file) on Quercus. We request you host the videos externally and add a link to them to save space on Quercus. 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 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.

## Required Preamble

The code you submit (`doodlejump.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, and a link to your video demonstration. This is necessary information for the TA to be able to mark your submission.

```
#####################################################################
#
# CSCB58 Fall 2020 Assembly Final Project
# University of Toronto, Scarborough
#
# Student: Name, Student Number
#
# Bitmap Display Configuration:
# - Unit width in pixels: 16
# - Unit height in pixels: 16
# - Display width in pixels: 512
# - Display height in pixels: 512
# - 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)
#
# Link to video demonstration for final submission:
# - (insert YouTube / MyMedia / other URL here).
#
# Any additional information that the TA needs to know:
# - (write here, if any)
#
#####################################################################
```

## Required Video Demonstration

You will need to include a short (5-10) video 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, ...)
2. Demonstrate the functionality of the additional features implemented for milestone #4 and #5 listed 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, if possible. UofT provides a media sharing service you can use to host the video, if you would rather not use Youtube. The URL should go in the preamble.

## 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.

Remember that *sharing your code* is also a violation, not just using someone else's. If you are using version control such as Git, ensure your repositories are private, and do not otherwise let anyone else see your code. It is fine to discuss ideas (and encouraged), but sharing code is forbidden.

## Useful Resources

MIPS System Calls Table
MIPS Reference Card
MIPS API Reference
Assembly Slides (UTSC)