Assembly Project: Breakout

Academic Integrity

All submissions will be checked for plagiarism. Make sure to maintain your academic integrity 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 risk much worse consequences by committing an academic offence.

Contents

1 Getting Started

For this project, you will be using MIPS assembly to implement a version of the popular retro game Breakout.

Since we do not have access to a physical computer that uses MIPS processors, we will be simulating this using MARS (or one of the other development tools that we provide). The MARS simulator not only simulates the processor but also a Bitmap Display and Keyboard input. If you have not already downloaded MARS, see the `Install and understand the software` page on Quercus and watch the recordings on Quercus that introduce MARS and assembly programming.

Read Section 2 to familiarize yourself with the game. Read Section 3 to familiarize yourself with the technical details of assembly and MARS that you need to know before getting started. Once you are ready to start, **download the starter files** and read Section 4 to see what is expected of your program and when. In addition to the example code discussed in Section 3, we provide a file `breakout.asm` with the beginnings of a game loop. It is this file where you begin to write your program..

1.1 Quick start: MARS

Running your code in MARS involves the following steps:

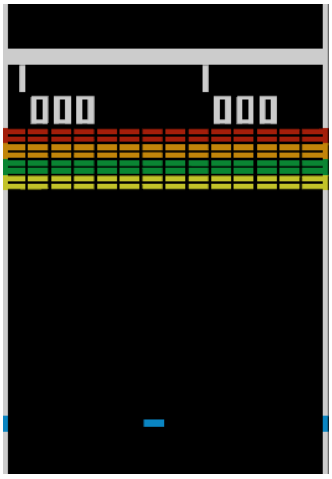
1. Withing MARS, open the starter file `breakout.asm`.
2. Set up the Bitmap Display by navigating to `Tools -> Bitmap Display`.
   2a. Configure the Bitmap Display. Remember to configure the base address.
   2b. Click `Connect to MIPS`, but don't close the window.
3. Setup the keyboard by navigating to `Tools -> Keyboard and Display MMIO Simulator`.
   3a. Click `Connect to MIPS`, but don't close the window.
4. Navigate to `Run -> Assemble`.

   Check for errors and inspect memory and its values for any bugs.
5. Navigate to `Run -> Go` to run your program.
6. In the keyboard area of your `Keyboard and Display MMIO` simulator, enter characters like `a, d, q`.

   Note: You will need to add code so that your program responds to these keys.

2 Breakout: The Game

The game Breakout has seen many version over the decades (like Araknoid and Brick Breaker), and is itself an extension of the classic game Pong. At its core, the game involves a paddle (controlled by the player), a ball, a set of bricks, and three walls on the left, right and top of the screen (see Figure 2.1). The goal of the game is to move the paddle left and right to bounce the ball toward the bricks at the top of the screen, which disappear when they are hit by the ball.



(a) Arcade

(b) Atari 2600

Figure 2.1: Screen captures of Breakout on different systems.

Notice that there is no wall at the bottom of Figure 2.1. If the ball falls off the bottom of the screen before the player can stop it with the paddle, that ball is lost and play continues with a new ball. The player typically starts with three balls, and the game ends when all three balls have fallen off the bottom edge of the screen.

To complete the level, the player must break all the bricks. This causes a new level to start, which typically increase in difficulty over time. For anybody who hasn't played this game before, you can try an online version of Atari Breakout at https://www.coolmathgames.com/0-atari-breakout or Arkanoid at https://www.crazygames.com/game/arkanoid.

Each version of Breakout makes modifications on this basic gameplay, by varying the speed, adding powerups, multiplayer settings, etc. For this project, you will be creating your own version, and your mark will reflect the difficulty of implementing the features you choose.

2.1 Game Controls

Your version of Breakout will use the keyboard keys `a` and `d` for moving the paddle. The `a` keyboard key moves the paddle to the left. The `d` keyboard key moves the paddle to the right. The paddle should not move 'up' nor 'down' (at least in the basic version of the game). When no key is being pressed by the user, then the paddle should not move.

You may need additional keys for other operations, such as starting the game, quitting the game,

resetting the game, etc. For the technical background on checking keyboard input, see Section 3.1.

2.2 The Bricks

There are many ways to track and draw the bricks in your version of Breakout. For example, in Figure 2.1a, bricks are separated by a small gap. But, in Figure 2.1b, there is no separation between any of the bricks. For the technical background on drawing bricks, see Section 3.2.

When the ball hits a brick, it should "break" (or start to break, depending on the features you implement). You should decide how many times the ball should hit the brick before the brick "fully breaks" (i.e., disappears from the screen). The basic game has the brick disappear when it has been hit once. If you decide that the brick should be hit more than once to disappear, you may also want to think about how a partially broken brick should look (compared to an unbroken brick). For instance, some versions change the colour of the bricks every time they are hit, or draw the brick with cracks in it. Refer to Section 4.2 for more information about this and other features.

2.3 Collision Detection

Part of what makes Breakout interesting is that the ball can collide with other objects (e.g., the paddle, a brick, a wall). When a collision occurs, the ball changes directions.

How the ball changes directions will be up to you, but for an enjoyable player experience, the bounce direction should be somewhat intuitive and reflect the behaviour that the player would expect. For example, if you choose to have the ball randomly change directions upon a collision, this would break the player's expectations and make the game unpleasantly difficult to play.

2.4 Game Over

When the ball collides with the bottom of the screen (where there is no wall), then the player loses that ball. The typical game allows the player to lose three balls before the game is completely over. On each attempt, the ball would be placed at a given starting point so the player can try again.

3 Technical Background

In addition to using MIPS assembly, there are three concepts you should be familiar with before starting the project:

1. Keyboard Input
2. Displaying Pixels
3. System Calls

Both Keyboard Input and Displaying Pixels use a concept called Memory Mapped I/O. This means that we can communicate with these peripherals as if they were in memory. Each peripheral (e.g., keyboard, bitmap display) has a corresponding memory address. Loading from, or storing to, that memory address (and nearby addresses) allows you to interface with the peripheral.

3.1 Keyboard Input

You must use the `Keyboard and Display MMIO Simulator` in MARS to support keyboard input. You can find it under the `Tools` menu in MARS. Once the window is open (Figure 3.1), you must also click `Connect to MIPS`. For step-by-step instructions on how to setup MARS, see Section 1.1.

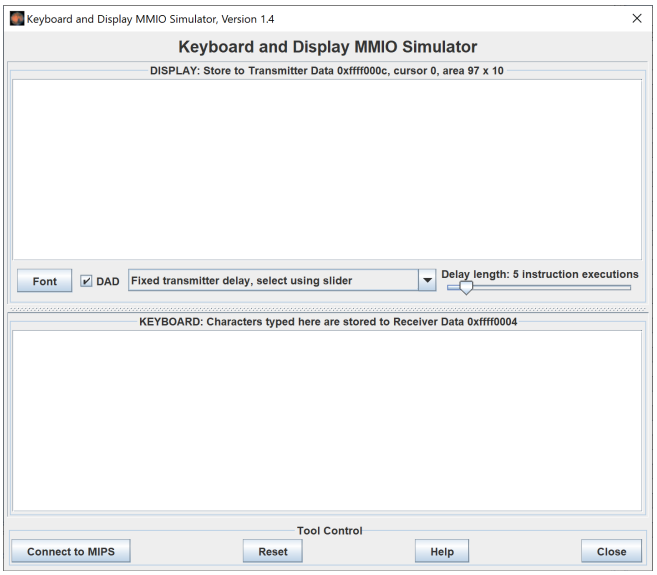


Figure 3.1: The Keyboard and Display MMIO Simulator

When a key is pressed, the processor will tell you by setting a location in memory (`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 address. If that memory address has a value of 1, then the key that was pressed is found in the next word in memory. The word contains the ASCII-encoded value of the key that was pressed. Listing 3.1 shows an excerpt of how this works in MIPS.

Listing 3.1: An excerpt of the keyboard.asm starter file

```
    .data
keyboard_address:     .word 0xffff0000

    # ...
    .text
    # ...

    lw $t0, keyboard_address        # $t0 = base address for keyboard
    lw $t8, 0($t0)                  # Load first word from keyboard
    beq $t8, 1, keyboard_input      # If first word 1, key is pressed

    # ...

keyboard_input:                     # A key is pressed
    lw $t2, 4($t0)                  # Load second word from keyboard
    beq $t2, 0x71, respond_to_Q     # Check if the key q was pressed

    # ...
```

3.2 Displaying Pixels

You must use the `Bitmap Display` in MARS to simulate the output of a display (i.e., screen, monitor). You can find it under the `Tools` menu in MARS. A bitmap display can be configured in many different ways (Figure 3.2); **make sure you configure the display properly before running your program.** Once the display is configured, you must also click `Connect to MIPS`. For step-by-step instructions on how to setup MARS, see Section 1.1.

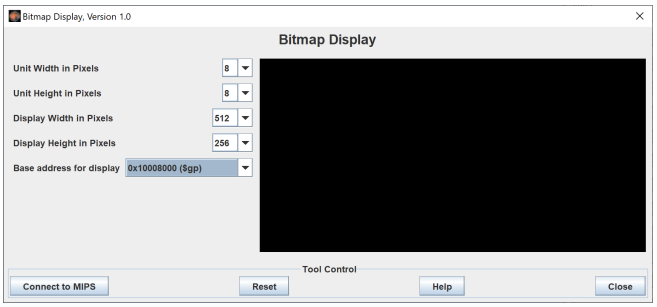


Figure 3.2: The Bitmap Display

The game will appear on the Bitmap Display in MARS. The display, visually, is a 2D array of "units", where each unit corresponds to a block of pixels. Here is how you can configure the display:

- The `Unit Width in Pixels` and `Unit Height in Pixels` is like a zoom factor. The values indicate how many pixels on the bitmap display are used to represent a single unit. For example, if you use 8 for both the width and height, then a single unit on the display would appear as an 8x8 box of a single colour on the display window.
- The `Display Width in Pixels` and `Display Height in Pixels` specified the width and height of the 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 units to draw based on those dimensions. For example, if your display width is 512 pixels and your unit width is 8 pixels, then your display is 64 units wide.
- The `Base address for display` indicates the location in memory that is used to display pixels on the screen.

Listing 3.2: An excerpt of the bitmap_display.asm starter file

```
    .data
display_address:     .word 0x10008000

    # ...
    .text
    # ...

    li $t1, 0xff0000          # $t1 = red
    li $t2, 0x00ff00          # $t2 = green
    li $t3, 0x0000ff          # $t3 = blue

    lw $t0, display_address   # $t0 = base address for display
    sw $t1, 0($t0)            # paint the first unit (i.e., top-left) red
    sw $t2, 4($t0)            # paint the second unit on the first row green
    sw $t3, 128($t0)          # paint the first unit on the second row blue

    # ...
```

You need to set the `Base address of display` 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 unexpected behaviour.

Memory is one-dimensional, but the screen is two-dimensional. Starting from the base address of the display, units are stored in a pattern called *row major order*:

- If you write a colour value in memory at the base address, a unit 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 unit of that colour one unit to the right of the first one.
- Once you run out of unit locations in the first row, the next unit value will be written into the first column of the next row, and so on.

Each pixel uses a 4-byte colour value, similar to the encoding used for pixels in Lab 7. In this case, the first byte is not 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). 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:

1. Calculate the colour code you want using the combination of red, green and blue components
2. Calculate the pixel location based on the display's width and height
3. Finally, store that colour value at the correct memory address

See Listing 3.2 for an example of how this looks.

3.3 System Calls

The `syscall` instruction is needed to perform special built-in operations. For example, we can use a syscall to sleep or exit the program gracefully. The syscall instruction looks for a number in register `$v0` and performs the operation corresponding to that value.

The sleep operation suspends the program for a given number of milliseconds. To invoke this operation, the value 32 is placed in `$v0` and the number of milliseconds to wait is placed in `$a0`. The listing below tells the processor to wait for 1 second before proceeding to the next line:

Listing 3.3: Invoking the sleep system call

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

To terminate a program gracefully, you do not need any arguments. The value to be placed in `$v0` now is 10. The listing below shows how to exit gracefully:

Listing 3.4: Invoking a system call to terminate the program

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

There is also a system call for producing random numbers. To generate a random number, you can either place 41 or 42 in `$v0`. In both cases, the argument `$a0` is used to indicate a random number generator ID (assuming you only use one random number generator, you can always use 0 here). When `$v0` is 41, the system call produces a random integer. But when `$v0` is 42, the system call produces a random integer up to a maximum value (exclusive). That maximum value must be provided in `$a1`. The listing below demonstrates how to generate a random number between 0 and 15:

Listing 3.5: Generate a random number between 0 and 15

```
li $v0, 42
li $a0, 0
li $a1, 16
syscall         # after this, the return value is in $a0
```

For those familiar with Java, MARS fulfills these system calls by using `Java.util.Random`. If you want deterministic random values, you will need to use another system call to set the seed of your random number generator. Refer to the MIPS System Calls reference (link on Quercus in the Project module).

4 Deliverables and Demonstrations

You may work in pairs on the project, or you can choose to work on your own. If you wish to work in pairs, you can work with your partner from the labs or work with someone different. The only rule is that your partner must still be from the same section as you (i.e. L0101, L0202, L5101).

You demonstrate your project twice:

1. The first project demonstration is in Week 11, where you are meant to demo Milestone 3. Failing to demonstrate Milestone 1 will result in a penalty of 20% of your overall project mark (meaning you can get a maximum of 12/15 on the project).

2. The second demonstration is in Week 12, where you demonstrate the finished project.

3. In both cases, you submit your files on Quercus before 6pm on the day of your lab session (just like in labs). Everybody needs to submit their files individually, even if you're working in pairs.

- **Deliverable 1**: Due on Quercus before 6pm on Monday March 27 (L0201), Wednesday March 29 (L0101), Thursday March 30 (L5101)
- **Demonstration 1**: During the lab session you are enrolled in (March 27, 29 or 30), 6pm-9pm
- **Deliverable 2**: Due on Quercus before 6pm on Monday April 3 (L0201), Wednesday April 5 (L0101), Thursday April 6 (L5101)
- **Demonstration 2**: During the lab session you are enrolled in (April 3, 5 or 6), 6pm-9pm

CAUTION

Your demonstrations are based on your deliverables. During the demonstration, expect that the file submitted on Quercus will be the one that we test. Make sure that it works before coming into the lab, because you will not have time to do more than a few minor bug fixes during the lab.

You must upload *every required file* for your deliverable submission to be complete. If you have questions about the submission process, please ask ahead of time. The required files for each deliverable are:

- Your project report: `project_report.tex`, `project_report.pdf` (as generated from the tex file)
- Your assembly code: `breakout.asm`

The project is divided into five milestones:

1. **Milestone 1:** Draw the scene (static; nothing moves yet) (e.g., as shown in Figure 2.1)
2. **Milestone 2:** Implement movement and other controls
3. **Milestone 3:** Collision detection
4. **Milestone 4:** Game features
5. **Milestone 5:** More game features

Each milestone is worth 4 marks, for a total of 12 marks for Demonstration 1 (assuming you complete Milestone 3) and 10 marks for the Final Demonstration (based on how much of Milestones 1-5 you complete, more details below). In Demonstration 1, the expectation is that you demonstrate a project that has reached Milestone 3. In the final demonstration, the expectation is that you demonstrate a project that has reached at least Milestone 4.

Milestones that have not been reached by the final demonstration receive a 0. A milestone has been reached when the code for that milestone is working *correctly* and the implementation is *non-trivial*. For example, a trivial implementation of Milestone 1 would be to have drawn only three bricks total. So it is possible to receive 1/2 marks for a milestone.

4.1 Preparing for Demonstration 1

Before Demonstration 1 (i.e. the in-lab component of Week 11), you must have completed milestones 1, 2, and 3. To receive full marks for each milestone, the TAs will be expecting the following (at minimum):

1. **Milestone 1:** Draw the scene (static; nothing moves yet) (e.g., as shown in Figure 2.1)

    a) Draw the three walls.
    b) Draw all the bricks. There should be at least three rows of bricks and at least three different coloured bricks (and a non-trivial number of bricks in each row).
    c) Draw the paddle.
    d) Draw the ball (at some initial location).

2. **Milestone 2:** Implement movement and other controls

    a) Move the paddle using keyboard input.
    b) Move the ball in any direction (no collisions yet).
    c) Re-paint the screen in a loop to visualize movement.
    d) Allow the player to quit the game.

3. **Milestone 3:** Implement collision detection

    a) The ball bounces when it hits another object (wall, brick, paddle).
    b) A brick disappears when it is hit by the ball once.
    c) The player loses when the ball reaches the bottom of the screen (where there is no wall).

To make this happen, consider the following steps:

1. Decide on how you will configure your bitmap display (i.e. the width and height in pixels).

    Include your configuration in the preamble of `breakout.asm`. Remember to also include your names and student numbers.
2. Decide on what will be stored in memory, and how this data will be laid out. Grid diagrams (i.e. using graph paper) are particularly useful when planning the elements of Milestone 1.

    Include this plan in your report.
3. Translate your plan into the `.data` section of your `breakout.asm` program. Assemble your program in MARS and inspect memory to ensure it matches your plan.

Include a screenshot (or multiple screenshots) of memory demonstrating that it has been laid out according to your plan.

4. Draw the scene (Milestone 1). Think carefully about functions that will help you accomplish this, and how they should be designed based on the variables you have in memory.

   Include a screenshot of the static scene in your report.

   Upload `breakout.asm` to Quercus so that you have a snapshot of your progress so far.
5. Implement movement and other controls (Milestone 2).

   Upload `breakout.asm` to Quercus so that you have a snapshot of your progress so far.
6. Decide on what should happen when the ball collides with an object.

   Include an answer to the question "How will the ball change directions when it collides?" in your report.
7. Implement collision detection (Milestone 3).

   Upload `breakout.asm` to Quercus so that you have a snapshot of your progress so far.

4.2 Preparing for the Final Demonstration

Before the Final Demonstration (i.e., before your 6pm lab time in Week 12), you should aim to complete Milestone 5 (or barring that, at least Milestone 4). These milestones are reached through a combination of easy features and hard features, as defined below.

4. **Milestone 4:** Game features (**one** of the combinations below)

   a) 4 easy features
   b) 2 easy features and 1 hard feature
   c) 2 to 3 hard features

5. **Milestone 5:** More game features (**one** of the combinations below)

   a) 7 easy features
   b) 5 easy features and 1 hard feature
   c) 3 easy features and 2 hard features
   d) 1 easy feature and 3 hard features
   e) 4 (or more) hard features

To earn these milestones, you should perform the following steps:

1. Implement an easy or hard feature.
2. Repeat the previous step until you have achieved your goal for Milestone 4 and/or 5.
3. Update your Demonstration 1 report based on any changes made.
4. Include a section in your report titled "How to Play". Include instructions for players based on the controls your game supports.

Easy Features

Easy features do not, typically, require significant changes to existing code or data structures. Instead, they are mostly "adding on" to your program. The easy features below are numbered so that you can refer to them by their number in the preamble.

1. Support "multiple lives" (at least 3) so that the player can continue the game multiple times. The state of the game (i.e., broken bricks) should be retained for subsequent attempts.

2. When the player has used up all their attempts (e.g., 1, 3, more than 3), display a Game Over screen. Restart the game if a "retry" option is chosen by the player. Retry should start a brand new game (no state is retained from previous attempts).

3. Support different speeds for the ball. That is, after a certain number of collisions, the ball not only changes direction by also increases in speed.

4. Add sound effects for different conditions like colliding with the wall, brick, and paddle and for winning and game over.

5. Allow the user to pause the game by pressing the keyboard key `p`.

6. Add a time limit to the game.

7. Add 'unbreakable' bricks.

8. Add a second paddle that is controlled by a second player using different keys.

9. Allow the player to launch the ball at the beginning of each attempt.

Hard Features

Hard features require more substantial changes to your code. This may be due to significant changes to existing code or adding a significant amount of new code. The hard features below are numbered so that you can refer to them by their number in the preamble.

1. Track and display the player's score, which is based on how many bricks have been broken so far.

2. Track and display the highest score so far (assumes that players can retry new games; see easy features).

3. When the ball bounces off the paddle, alter the angle based on which side of the paddle the ball comes in contact with. Hitting the middle of the paddle makes the ball rebound straight up, whereas the bounce angle is more pronounced to the left or the right, depending on how far the ball makes contact to the left or the right side of the paddle

4. Require bricks be hit by the ball multiple times before breaking. Demonstrate the players progress toward breaking a brick in some way (e.g., different colours).

5. Create a second level with a different layout of bricks.

6. Create menu screens for things like level selection, a score board of high scores, etc (assumes you have completed at least one of those hard features).

7. Add some animation to, for example, your bricks when they break/dissappear.

8. Add a powerup of some kind that is activated on certain conditions (e.g., a certain brick is broken, 10 bricks are broken without losing a ball, items fall from the sky to be caught by the paddle etc.)

   Each additional powerup would be its own easy or hard feature. For example, Arkanoid has a powerup that allows the paddle to 'shoot' bricks directly above it by pressing the space bar. Another powerup it supports is splitting the ball into multiple balls (at least three) that move at different angles, where your current attempt doesn't end until all of the balls have gotten past your paddle. These are examples of hard powerups you can add.

   Arkanoid also has powerups that make the paddle longer, give the player an additional life, allow the paddle to "catch" the ball or skip the player immediately to the next level. These would be easy features to add. See the guide to Easy Features and Hard Features to figure out what recognition you'll receive for the powerups you create.

4.3 Advice

Once your code starts, it should have a central processing loop that does the following (the exact order may change, but this is a good reference):

1. Check for keyboard input
2. Check for collision events
3. Update locations (ball, paddle)
4. Redraw the screen
5. Sleep.
6. 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 cannot 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 is not 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.

Here are some general tips:

1. **Get a piece of graph paper.** You need to plan how big your walls, bricks, paddles, and ball will be. Let every square on your graph paper represent a location that the ball can occupy in the game. Figure out where everything should be for Milestone 1. You might need to change your bitmap display settings to fit your design.

2. **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.

3. **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.

4. **Create meaningful labels.** Meaningful labels for variables, functions and branch targets will make your code much easier to debug.

5. **Write comments.** Without useful 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.

6. **Start small.** Do not 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.

7. **Play the game.** There are many clones of Breakout on the internet (e.g., search itch.io), and quite a few are playable in the browser. Get a sense of the core gameplay by playing one of these clones.