Assembly Project: Tetris

Academic Integrity

All submissions will be checked for plagiarism, and for AI-generated content. 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 Tetris.

Since we do not have access to a physical computer that uses MIPS processors, you will be creating and simulating your game using Saturn or MARS (or any other development tools that we provide). MARS and Saturn not only simulate the main processor but also a Bitmap Display and Keyboard input. If you have not already downloaded one of these simulators, see the `Assembly Language Simulators` 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 the simulators before getting started. Once you are ready to start, **download the starter files** and read Section 4 to see what is expected of you and when. In addition to the example code discussed in Section 3, we provide a file `tetris.asm` with the beginnings of a game loop. It is this file that you will fill in to complete your assembly program.

1.1 Quick start: Saturn & MARS

Saturn and MARS are the two simulators that we support in this course:

1. MARS is an IDE for MIPS assembly language, created by Missouri State University (`https://courses.missouristate.edu/kenvollmar/mars/`). We were using it for several years and it offers several interesting features that have been added and refined over time. That being said, it's not supported as much anymore, despite some known bugs.

2. Saturn is a similar IDE that was developed by Taylor Whatley, a former student and current TA of the course. As far as the project is concerned, it has all the features and capabilities of MARS, while also fixing all the bugs that MARS was known for. We are continuing to work on it and add new features, so please give us feedback on behaviours you notice or features that you'd want.

The download links for both of these IDEs can be found on Quercus.

1.1.1 Opening and Running Programs in Saturn & MARS

1. Within Saturn, open the starter file `tetris.asm`. You can do this by dragging `tetris.asm` into the Saturn window.

2. Press Ctrl + T (or Cmd + T on macOS) to open the terminal. Navigate to the Bitmap tab.

   a) Configure the Bitmap Display. Remember to configure the base address.

   b) To send your keystrokes to your MIPS window, click the bitmap window on the left. Your keystrokes will be sent to your MIPS assembly app as long as this window is in focus.

3. Click the green "Play" button on the top right corner to assemble and run your app. Check for any errors in the Console tab.

4. While the Bitmap Display is selected on Bitmap tab, enter characters like `a, d, q`.

If you'd prefer to run your code in MARS, use the following steps:

1. Withing MARS, open the starter file `tetris.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`.

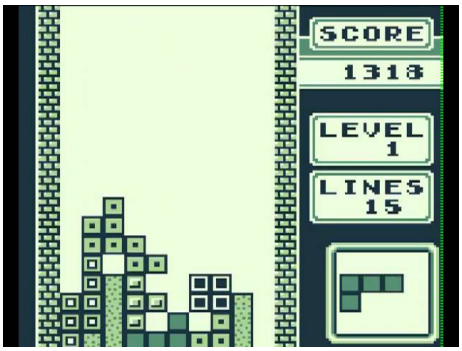
Note: Regardless of whether you use MARS or Saturn, you will need to add code so that your program responds to these keyboard inputs.

2 Tetris: The Game

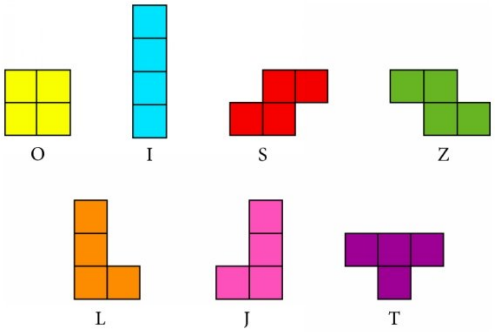
Tetris is an arcade puzzle game that was created in 1985 by Alexey Pajitnov. It is one of the simplest and most well-known games that has seen many versions over the decades.

The core game involves dropping shapes made of different configurations of 4 blocks (called tetrominoes) into the bottom of a vertical playing field. When a tetromino lands on the bottom of the field or on another piece, it becomes fixed in place and a new tetromino appears at the top of the field. The player can rotate the current tetromino, move it left and right within the playing field, and drop it down (see Figure 2.1a).

If placing a tetromino completes a horizontal line of blocks across the field, that line of blocks is removed and the rows of blocks above drop down one line. The goal of the game is to keep filling rows and avoid having the playing field fill up vertically with blocks, which would end the game.



(a) Tetris on Gameboy (1989)



(b) Tetrominoes

Figure 2.1: Screen captures of Tetris and the Tetrominoes.

If you haven't seen or played this game before, you can try an online version of Tetris here: `https://jstris.jezevec10.com/` or on tetr.io (better for playing against another players): `https://tetr.io/`.

Each version of Tetris has made modifications on this basic gameplay by introducing levels, varying the speed, adding powerups, animations, 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.

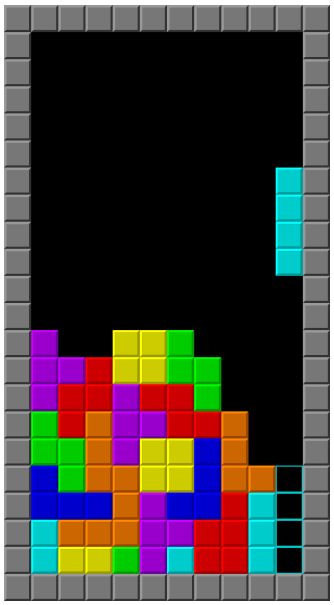


Figure 2.2: Arcade Tetris

2.1 Game Controls

Your implementation of Tetris will use the keyboard keys `w`, `a`, `s` and `d` for moving and rotating the tetromino pieces. The `a` key moves the piece to the left. The `d` key moves the piece to the right. The `w` key rotates the piece by 90 degrees (usually clockwise). The `s` key moves the piece toward the bottom of the playing field (either one line at a time or all at once, that's your choice). If no key is pressed by the player, then the piece does not move (at least in the basic version).

You may need additional keys for other operations, such as starting the game, quitting the game, pausing the game, resetting the game, etc. For the technical background on checking keyboard input, see Section 3.1.

2.2 The Tetrominoes

The most important design decisions you'll need to make are how to store/draw the tetromino pieces, and how to decide when a piece has collided with another piece (or the bottom of the playing field).

As seen in Figure 2.1b, there are seven types of tetrominoes, each with a single-letter name. For Milestone 3, you only need to implement one of these (anything but the 2x2 'O' piece). Typically, a piece will start in its default orientation, and will stay in its current location when you rotate it 90 degrees with the `w` key. This means you'll want to store (likely in memory) the current orientation of the piece and the current X and Y location of the piece on the playing field. You'll use those details in your tetromino drawing function to draw the blocks of that piece on the screen.

For the technical background on drawing blocks, see Section 3.2.

As you progress onto later milestones, you'll also want to store other information (such as the colour of the piece), so it's good to plan ahead for that as you develop your early milestones. Refer to Section 4.2 for more information about this and other features.

One issue you'll need to check is that you don't rotate your tetromino into a location that's already occupied by an existing piece. That can be handled in multiple ways, such as shifting the tetromino to

the left or the right by one space before drawing it, or not rotating it at all. Either way, you'll need a collision detection function to determine if this condition takes place.

2.3 Collision Detection

The other challenging task in Tetris is handling the case where a tetromino piece drops down onto one of the existing pieces, or onto the bottom of the playing field. The way Tetris is implemented, if any of the four blocks in the player's tetromino piece collides vertically with any block of an existing piece, the player's tetronimo is fixed in place at that location and a new tetromino is generated at the top of the field. This means that you need to store the collection of pieces that have already been placed, including the 'spaces' that are not occupied by a piece.

You'll also need a way to detect if the current tetromino collides with anything when the player tries to move it left or right. If there is a horizontal collision, the player's tetromino won't move in that direction, but it also doesn't get fixed in place. The player can still move the piece after horizontal collisions with other objects.

Storing the playing field and checking for collisions can also be implemented in multiple ways. You can store a representation of the playing field as a grid of occupied or unoccupied spaced and then have a function that draws the game from this stored representation. Or just store the raw pixels and alter that as the game is played. Either way, you'll want a function that checks for collisions with neighbouring spaces, both horizontally and vertically.

2.4 Game Over

When there is no room left at the top of the playing field to generate a new tetromino, this triggers the 'game over' condition and the game ends. You can decide whether to have the game halt or restart at this point, or something more advanced like displaying a 'game over' message (as one of your features for Milestone 4 or 5).

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

3.1.1 Saturn keyboard input

From your main development window, Press Ctrl + T (or Cmd + T on macOS) to open the terminal. Navigate to the Bitmap tab, which will display the bitmap window and settings. To send your keystrokes to your MIPS window, click on the bitmap window on the left. Your keystrokes will be sent to your MIPS assembly app as long as this window is in focus.

Note: Following these instructions allows the MIPS window to recognize your keystrokes. You still need to add key handling code to your program for it to respond to these keystrokes.

3.1.2 MARS keyboard input

If you are using MARS, you will need to use the `Keyboard and Display MMIO Simulator` 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 set up MARS, see Section 1.1.

3.1.3 Handling Keystroke Events

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 your program won't know that a key has been pressed until you check the contents of that memory address for a new keystroke event (an act known as polling). If that memory address has a value of 1, then a key has been pressed since the last time you checked. The ASCII-encoded value of the key that was pressed is found in the next word in memory (`0xffff0004`). Listing 3.1 shows an excerpt of how this works in MIPS.

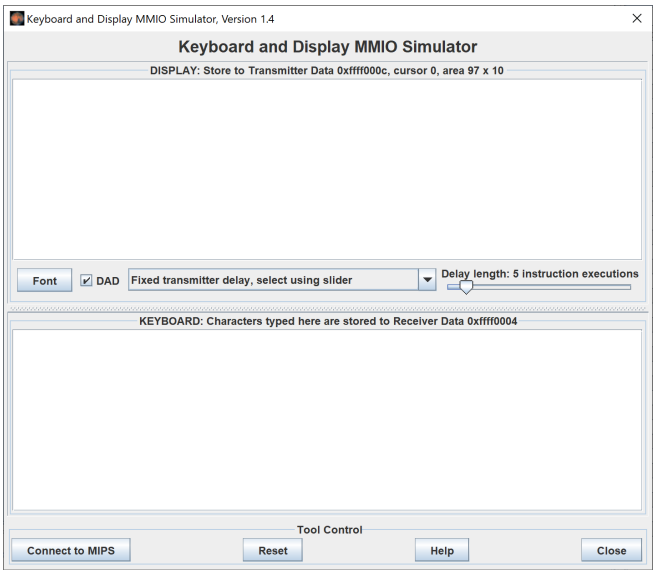


Figure 3.1: The Keyboard and Display MMIO Simulator in MARS

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

3.2.1 Displaying pixels in Saturn

If you are using Saturn, you can view the Bitmap Display by pressing Ctrl + T (or Cmd + T on macOS) to open the terminal. Navigate to the Bitmap tab and configure the Bitmap Display to the dimensions you want for your game. Remember to configure the base address.

When you run your game, clicking on the bitmap window on the left will allow you to send keystroke inputs to your MIPS window.

3.2.2 Displaying pixels in MARS

If you are using MARS, 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.

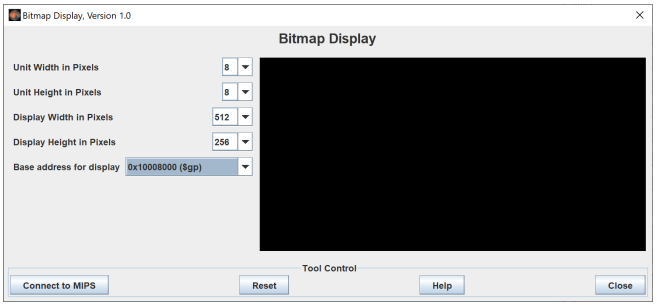


Figure 3.2: The Bitmap Display in MARS

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.

3.2.3 Drawing to the Bitmap

Regardless of the IDE you choose, tou need to set the `base address of display` to memory location `0x10008000`. 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 is left as another value (i.e. the default in MARS is the "static data" location), the bitmap display will look for pixel values in the wrong section of memory, which may cause unexpected behaviour.

You can set the dimensions of your bitmap display to whatever would suit your game the best. If your bitmap exceeds a certain size (∼8000 pixels), the bitmap region in memory will start to overlap with the `.data` section of memory. You can fix this by allocating space to a variable in the first line of the `.data` section of memory and then never using that variable. Talk to one of the instructors if you'd like an explanation of what that means and why that works.

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.

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

    # ...
```

- 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, both Saturn and MARS fulfill 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, your partner does not have to be the same partner you had for the labs. However, we ask that your partner be from the same lab day as you.

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 25 (L0101), Wednesday March 27 (L0201), Thursday March 28 (L5101)
- **Demonstration 1**: During the lab session you are enrolled in (March 25, 27 or 28), 6pm-9pm
- **Deliverable 2**: Due on Quercus before 6pm on Monday April 1 (L0101), Wednesday April 3 (L0201), Thursday April 4 (L5101)
- **Demonstration 2**: During the lab session you are enrolled in (April 1, 3 or 4), 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: `tetris.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 3 marks, for a total of 9 marks for Demonstration 1 (assuming you complete Milestone 3) and 15 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, implementing Milestone 3 with the 2x2 tetromino would trivialize the rotation task. So it is possible to receive part marks for a milestone, if the TA deems the task too easy.

4.1 Preparing for Demonstration 1

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

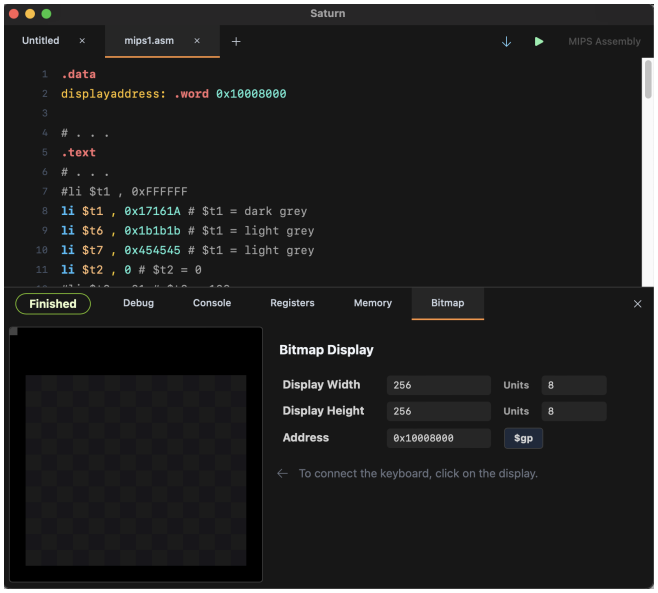


Figure 4.1: Drawing the Tetris background

1. **Milestone 1:** Draw the scene (static; nothing moves yet):

   a) Draw the three walls of the playing area.

   b) Within the playing area, draw a grid background that shows where the blocks of each tetromino will be aligned (e.g., similar to the checkerboard grid in Figure 4.1).

   c) Draw the first tetromino (at some initial location).

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

   a) Move the tetromino in response to the `W`, `A`, `S` and `D` keys (to make the tetromino move left and right, rotate and drop).

   b) Re-paint the screen in a loop to visualize movement.

   c) Allow the player to quit the game.

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

a) When the tetromino moves against the left or right side wall of the playing area, keep it in the same location.

b) If the tetromino lands on top of another piece or on the bottom of the playing area, leave it there and generate a new piece at the top of the playing area.

c) Remove any lines of blocks that result from dropping a piece into the playing area.

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 `tetris.asm`. Remember to also include your name(s) and student number(s).
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, and submit it on Quercus.
3. Translate any sprites or pixel grids from your plan into the `.data` section of your `tetris.asm` program. Assemble your program in Saturn or MARS and inspect memory to ensure it matches your plan.

   Submit 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 this static scene in your report.

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

   Upload `tetris.asm` to Quercus so that you have a snapshot of your progress so far.
6. Decide on what should happen when the tetromino collides with an object.
7. Implement collision detection (Milestone 3).

   Upload `tetris.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) 5 easy features

   b) 3 easy features and 1 hard feature

   c) 1 easy feature and 2 hard features

   d) 3 hard features

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

   a) 8 easy features
   b) 6 easy features and 1 hard feature
   c) 4 easy features and 2 hard features
   d) 2 easy feature and 3 hard features
   e) 4 (or more) hard features

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

1. Save a working copy of your game (in case the new feature breaks something).
2. Implement an additional easy or hard feature to your game.
3. Repeat the previous step until you have achieved your goal for Milestone 4 and/or 5.
4. Update your Demonstration 1 report based on any changes made.
5. 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. Implement gravity, so that each second that passes will automatically move the tetromino down one row.

2. Assuming that gravity has been implemented, have the speed of gravity increase gradually over time, or after the player completes a certain number of rows.

3. When the player has reached the "game over" condition, display a Game Over screen in pixels on the 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).

4. Add sound effects for different conditions like rotating and dropping tetrominoes, and for winning and game over.

5. If the player presses the keyboard key `p`, display a "Paused" message on screen until they press `p` a second time, at which point the original game will resume.

6. Add levels to the game that trigger after the player completed a certain number of rows, where the next level is more difficult in some way than the previous one.

7. Start the level with 5 random unfinished rows on the bottom of the playing field.

8. Show an outline of where the piece will end up if you drop it (see Figure 2.2).

9. Add a second playing field that is controlled by a second player using different keys.

10. Have a panel on the side that displays a preview of the next tetromino that will appear (see Figure 2.1a).

11. Assuming that you've implemented the score feature (see the hard features) and the ability to start a new game (see easy features), track and display the highest score so far. This score needs to be displayed in pixels, not on the console display.

12. Assuming that you've implemented the full set of tetrominoes, make sure that each tetromino type is a different colour.

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 lines have been completed so far. This score needs to be displayed in pixels, not on the console display.

2. Implement the full set of tetrominoes.

3. 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).

4. Add some animation to lines when they are completed (e.g. make them go poof).

5. Play the Tetris theme music (aka "Korobeiniki") in the background while playing the game.

6. Have special blocks randomly occur in some tetrominoes that do something special when they are in a completed line (e.g. they destroy the line above and below as well).

7. Add a powerup of some kind that is activated on certain conditions (e.g., when you complete 4 rows at once, when you complete 20 rows). Each powerup would be its own easy or hard feature and would be classified according to the TA's discretion.

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 tetromino location / orientation
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. Simulated MIPS processors are not typically 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 assembly programming tips:

1. **Get a piece of graph paper.** Let every square on your graph paper represent the space that a single block of a tetromino can occupy in the game. Use the grid of the graph paper to plan how big your walls, playing field and tetrominoes will be. Decide how many bitmap units will go into a single square in your graph paper. Figure out where everything should be for Milestone 1. You might need to change your bitmap display settings to fit your design.

2. **Measure twice, cut once.** It's well worth spending time coming up with a good memory layout because a bad or overly complex system turns into a lot of extra assembly code gameplay.

3. **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.

4. **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.

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

6. **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.

7. **Start small.** Do not try to implement your whole game at once.

8. **Use breakpoints for debugging.** 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. Use breakpoints and poke around on the registers tab to diagnose a problem by checking if the values are what you expect. In Saturn you can breakpoint a troublesome instruction and step backwards to see how your instructions created an unexpected result.

Here are some tips that are specific to the Tetris game:

1. **Storing the current tetromino.** Each tetromino has a different shape, and rotations can lead to 1-4 versions of each shape. Don't try storing these in registers. It's better to store each tetromino shape in memory (consider 4x4 blocks) and then refer to the memory location of a particular shape when drawing it on the bitmap display.

2. **Storing the past tetrominoes.** It's a good idea to have the contents of the playing field stored in memory, separate from where the player's current tetromino is stored. It'll make it easier to draw those rows and detect collisions.

3. **Check for collisions betore drawing the tetromino.** To know if the current tetromino is allowed to move in response to keyboard input, you need to check for collisions. That means looking at the four blocks that make up each tetromino and checking if the position underneath each block is empty (if the player is trying to move down) or if it's against the wall (if the player is moving to the side). It's a good idea to handle each of these collision checks in its own function call.

4. **Play the game.** Try some of the links we provide to play examples of the game, to get a sense of the core gameplay.