Assembly Project – CSCB58 Summer 2024

Tetris Game

1. Overview

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 MARS). MARS not only simulate the main processor but also a Bitmap Display and Keyboard input.

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

2.1 Important Info and Due Dates

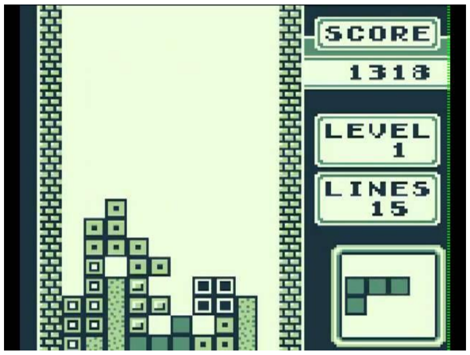
- **Check Piazza frequently for FAQs and updates.**
- **Checkpoint Demonstration**: Due during your usual **TA interview** slot on **Week of July 29th**.
- **Final Submission**: Due **Wednesday, August 7th, 2024**, 11:59pm (on Quercus).

2. Tetris: The Game

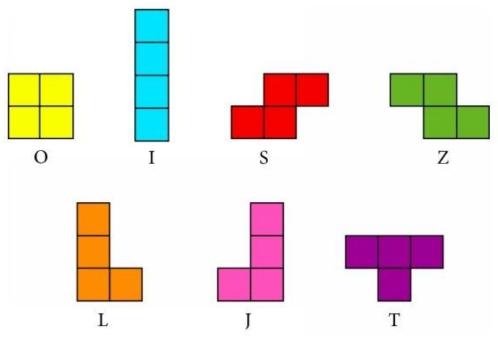
Tetris is an arcade puzzle game that was created in 1985 by Alexey Pajitnov. It is one of the simplest and most well-known games that has seen many versions over the decades.

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: Jstris (jezevec10.com) or on TETR.IO (better for playing against another players).

Each version of Tetris has made modifications to 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.

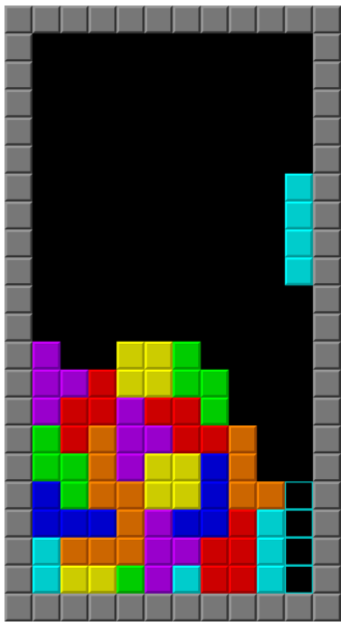


Figure 2.2: Arcade Tetris

2.2 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.

2.3 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.

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 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.

2.4 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 tetromino 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.

2.5 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

You will create this game using the MIPS assembly language taught in class and MARS. However, there are a few concepts that we will need to cover in more depth here to prepare you for the project: displaying pixels, taking keyboard input and system calls (`syscall`).

3.1 Displaying Pixels Using a Framebuffer

To display things on the screen, we will use a *framebuffer*: an area in memory shown as a 2D array of "units", where each "unit" is a small box of pixels of a single colour. Figure 3.1 on the right shows a framebuffer of width 10 and height 8 units.

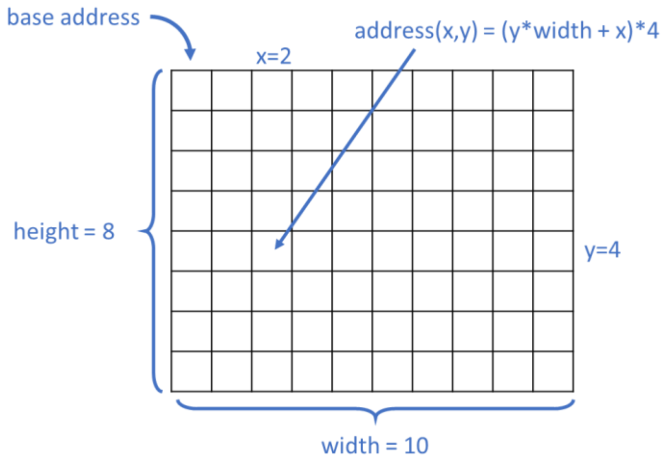


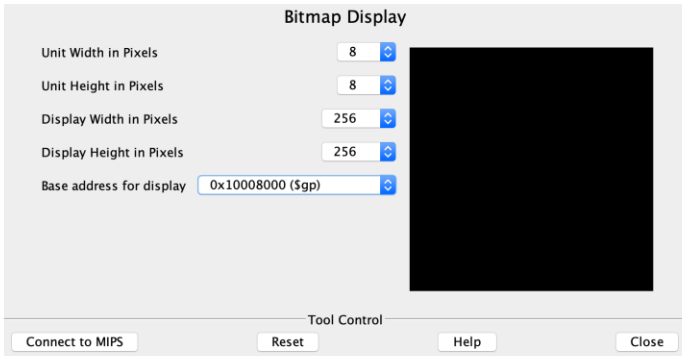
Figure 3.1: Framebuffer

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

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

The MARS Bitmap Display

MARS allows us to map a framebuffer in memory to pixels on the screen using the **Bitmap Display** tool, which you launch by selecting it in the MARS menu: **Tools → Bitmap Display**. The Bitmap Display window is shown on the right.



Bitmap Display

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)

To use the Bitmap Display, you need to:

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

Tip: We recommend using `0x10008000` as the base address for your framebuffer, but only if your framebuffer has 8192 cells or fewer, which is enough for 128x64 or 64x128 (if you want a larger framebuffer, ask us on Piazza; you'll have to add a variable at the beginning of the .data section). When using the Bitmap Display, make sure to change the "base location of display" field. If you set it to the default value *(`static data`)* provided by the Bitmap Display dialog, this will refer to the "*`.data`*" section of memory and may cause the display data you write to overlap with instructions that define your program, leading to unexpected bugs.

Bitmap Display Starter Code

To get you started, the code below provides a short demo, painting three units at different locations with different colours. Understand this demo and make it work in MARS.

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

```
# Bitmap display starter code
#
# Bitmap Display Configuration:
# - Unit width in pixels: 4
# - Unit height in pixels: 4
# - Display width in pixels: 256
# - Display height in pixels: 256
# - Base Address for Display: 0x10008000 ($gp)
#
.eqv  BASE_ADDRESS            0x10008000

.text
      li $t0, BASE_ADDRESS    # $t0 stores the base address for display
      li $t1, 0xff0000        # $t1 stores the red colour code
      li $t2, 0x00ff00        # $t2 stores the green colour code
      li $t3, 0x0000ff        # $t3 stores the blue colour code
```

```
    sw $t1, 0($t0)      # paint the first (top-left) unit red.
    sw $t2, 4($t0)      # paint the second unit on the first row green. Why $t0+4?
    sw $t3, 256($t0)    # paint the first unit on the second row blue. Why +256?

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

3.2 Keyboard

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

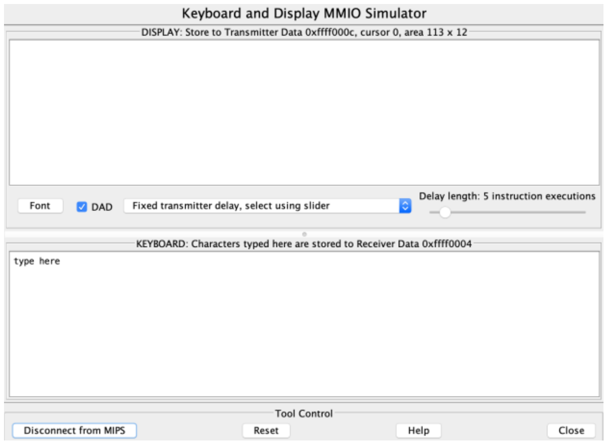

Keyboard and Display MMIO Simulator
DISPLAY: Store to Transmitter Data 0xffff000c, cursor 0, area 113 x 12
Font
DAD
Fixed transmitter delay, select using slider
Delay length: 5 instruction executions
KEYBOARD: Characters typed here are stored to Receiver Data 0xffff0004
type here
Tool Control
Disconnect from MIPS
Reset
Help
Close

Figure 2: MARS Keyboard and MMIO Simulator

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

Note the MARS simulator does not support pressing multiple keys at the same time, which you may need to think about when implementing jumps to left and right.

Fetching Keyboard Input

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

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

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

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

3.3 Useful Syscalls

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

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

To do this, put the value 41 or 42 into register `$v0`, then put the ID of the random number generator you want to use into `$a0` (since we're only using one random number generator, just use the value `0` here). If you selected service 42, you also have to enter the maximum value for this random integer into `$a1`. Once the syscall instruction is complete, the pseudo-random number will be in `$a0`.

```
li $v0, 42
li $a0, 0
li $a1, 28
syscall
```

The other syscall service you will want to use is the **sleep operation**, which suspends the program for a given number of milliseconds. To invoke this service, the value `32` is placed in `$v0` and the number of milliseconds to wait is placed in `$a0`:

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

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

4. Your Code

4.1 Getting Started

You should work in pairs on the project, or you can choose to work on your own. Your partner should be the same partner you had for the labs. Keep in mind that you will be called upon to explain your implementation to your TAs when you demo your game.

You will create an assembly program named `tetris.asm`. You'll design your program from scratch, but you must begin your file with the preamble starter code we include below.

1. Open the provided `tetris.asm` file provided.
2. Set up display: Tools > Bitmap display (this demo shows how to do this)
    - Set parameters like unit width and height (we recommend 8) and base address for display (we recommend `0x10008000`).
    - Click "Connect to MIPS" once these are set.
3. Set up keyboard: Tools > Keyboard and Display MMIO Simulator
    - Click "Connect to MIPS"

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

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

4.2 Required Preamble

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

```
######################################################################
# CSCB58 Summer 2024 Assembly Final Project - UTSC
# Student1: Name, Student Number, UTorID, official email
# Student2: Name, Student Number, UTorID, official email
#
# Bitmap Display Configuration:
# - Unit width in pixels: 4 (update this as needed)
# - Unit height in pixels: 4 (update this as needed)
# - Display width in pixels: 256 (update this as needed)
# - Display height in pixels: 256 (update this as needed)
# - Base Address for Display: 0x10008000 ($gp)
#
# Which milestones have been reached in this submission?
# (See the assignment handout for descriptions of the milestones)
# - Milestone 1/2/3/4/5 (choose the one the applies)
#
# Which approved features have been implemented? "use numbers from the handout"
# (See the assignment handout for the list of features)
# Easy Features:
# 1. (fill in the feature, if any)
# 2. (fill in the feature, if any)
# ... (add more if necessary)
# Hard Features:
# 1. (fill in the feature, if any)
# 2. (fill in the feature, if any)
# ... (add more if necessary)
# How to play:
# (Include any instructions)
# Link to video demonstration for final submission:
# - (insert YouTube / MyMedia / other URL here). Make sure we can view it!
#
# Are you OK with us sharing the video with people outside course staff?
# - yes / no
#
# Any additional information that the TA needs to know:
# - (write here, if any)
#
######################################################################
```

5. Deliverables and Demonstrations

The project is divided into five milestones:

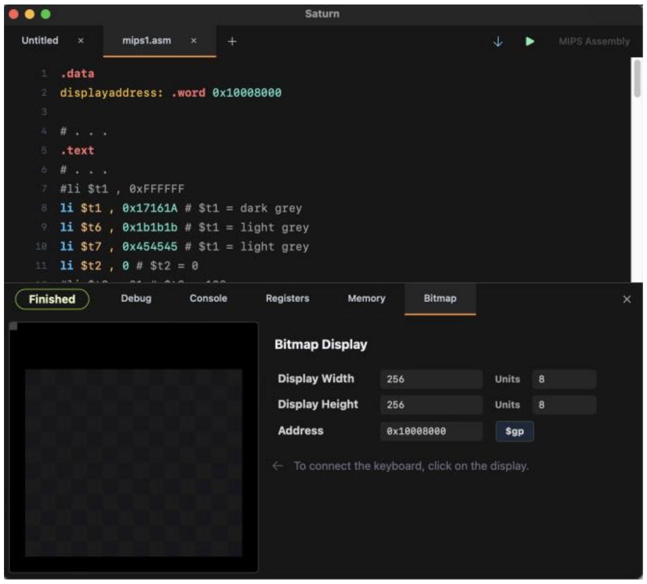
1. **Milestone 1:** Draw the scene (static; nothing moves yet) (e.g., as shown in Figure 2.1)
2. **Milestone 2:** Implement movement and other controls
3. **Milestone 3:** Collision detection
4. **Milestone 4:** Game features
5. **Milestone 5:** More game features

Each milestone is worth 4 marks, for a total of 12 marks for Demonstration 1 (assuming you complete Milestone 3) and 20 marks for the Final Demonstration (based on how many 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 your complete project (up to milestone 5).

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.

5.1 Preparing for Demonstration 1

Before Demonstration 1, 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):



```
.data
displayaddress: .word 0x10008000

# . . .
.text
# . . .
#li $t1 , 0xFFFFFF
li $t1 , 0x17161A # $t1 = dark grey
li $t6 , 0x1b1b1b # $t1 = light grey
li $t7 , 0x454545 # $t1 = light grey
li $t2 , 0 # $t2 = 0
```

Bitmap Display

Display Width	256	Units	8
Display Height	256	Units	8
Address	0x10008000	$gp	

To connect the keyboard, click on the display.

Figure 5.1: 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 5.1).

    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.

    Have this plan ready during your demonstration.

3. Translate any sprites or pixel grids from your plan into the .data section of your tetris.asm program. Assemble your program in MARS and inspect memory to ensure it matches your plan.

    Check memory to make sure 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.

    Save a screenshot of this static scene for your demonstration.

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.

5.2 Preparing for the Final Demonstration

Before the Final Demonstration, 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)

   Note that the combinations below denote the total number of features in your game.

   a) 8 easy features
   b) 6 easy features and 1 hard feature
   c) 4 easy features and 2 hard features
   d) 2 easy features and 3 hard features
   e) 1 easy feature and 4 hard features
   f) 5 (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. Include a section in your preamble 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 are triggered 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. 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.
11. Assuming that you've implemented the full set of Tetrominoes, make sure that each tetromino type is a different colour.
12. Have a panel on the side that displays a preview of the next tetromino that will appear (see Figure 2.1a).
13. Assuming that you've implemented the previous feature, extend it to show a preview of the next 4-5 pieces.
14. Implement the "save" feature, where you can save the current piece on the side instead of playing it. The game would skip to the next piece, and then allow you to retrieve the saved piece later in the game.

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.
8. Implement the wall kick feature of SRS, when rotating pieces: https://harddrop.com/wiki/SRS. This approach to rotation checks various alternate locations if the rotation would cause the current piece to intersect with the walls or other pieces.

5.2 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 copying 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 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.

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.

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 before 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.

5.3 Checkpoint Demo (Demonstration 1)

In the last week of classes (July 29th / August 1st), there will be a checkpoint session with your TA in your usual designated slot. By this point, you are expected to have **completed** the first three milestones at least and have started working on some of the features from milestone four. You will need to demonstrate your running code, and we will briefly ask you about how you have chosen to design certain parts of your game, and what your plan is for completing it. You need to submit the code for the checkpoint.

5.4 Final Submission

The deadline for final submission is going to be the last day of classes (August 7th). As such, we won't have an opportunity to meet to discuss your project. To help us evaluate your final project (so we don't miss anything), you need to submit a short video walking us through your project. ***Any final submission that does not include a video will not be marked***. If you have concerns with this, please get in touch with us *immediately.*

You will submit your `tetris.asm` (and only this one file) to Quercus. You will host the videos externally (on your OneDrive, YouTube, Google Drive, etc.) and add a link to it in your submission. Look below for additional details on what to include in the file and the video. You can submit the same file multiple times and only the latest version before the deadline will be marked. It is also a good idea to backup your code after completing each milestone or additional feature, to avoid the possibility that the completed work gets broken by later work. Again, make sure your code has the required preamble as specified above.

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

5.5 Required Video Demonstration (Demonstration 2)

You will need to include a short video (aim for 5 minutes, and no more than 7) 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 outlined in milestones 1 to 3 briefly.
2. Demonstrate the functionality of the additional features implemented for milestones 4 and 5 (remember to also list them in the preamble).
3. If you were unable to complete all milestones, explain what difficulties you encountered and show what progress you had on those features.
4. Tell us any other information you think would be useful to us while we are evaluating your work.

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

For sharing the video with us you will send a URL (in the preamble). U of T provides a media sharing service you can use to host the video, or you can use YouTube and similar services of your choice. Another option is Office 365, provided by the university, also includes OneDrive which allows sharing videos as well. It's up to you! Regardless of what service you use, make sure to include the URL in the preamble of the submitted file, and **make sure the video is viewable by the course staff**.

5.6 Marking

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

5.7 Bonus for Polish

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

6. Academic Integrity

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

Please note that all submissions will be checked for plagiarism. Make sure to maintain your academic integrity carefully and protect your own work. It is much better to take the hit on a lower assignment mark (just submit something functional, even if incomplete), than risking much worse consequences by committing an academic offence.

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

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

7. Useful Resources

- Assembly slides on Quercus
- Quercus assembly resource page
- MIPS System Calls Table
- Tools for editing sprites and pixel art.