Pong in Mips

# Writeup for end users

Preamble

Pong is one of the oldest video games ever made, and is probably one of the most well known original video games. It was released in 1972, and it was originally made on a purely hardware level, with logic gates and other electronic components, because microcontrollers were more expensive at the time.  
The game is an electronic version of ping pong, where the player, and a computer player, must try and keep a ball from going into their respective “goals,” while also trying to get the ball into their opponents goal. Points are awarded to players when the ball goes into the opponent’s goal. Whoever has the most points (or gets to x points first) wins.

Summary

This version of Pong has been made as modular as possible. The end user can customize almost all aspects of the game, and there are even some things behind the scenes that can be configured too. The end user isn’t able to customize as much due to not everything being tested.  
Excluding the exception handler, I wrote ~1500 lines for this project, with maybe ⅓-¼ of that coming from past projects.

Configuration Details

To run this game, all the user must do is:

1. Open the project in Mars
2. Navigate to the “Tools” column, and open “Bitmap Display” and “Keyboard and Display MMIO Simulator”
3. In the Bitmap Display, set the display height and width to 512, the unit height/width to 4, and the base address to “0x10040000 (heap)”
   1. The display can be set in any way such that display÷unit=128, and the aspect ratio is 1:1 for the unit size and display size
4. Press “Connect to MIPS” on both the Bitmap Display and the Keyboard and Display MMIO Simulator
5. Build and run the program

UI,rules, goals

* UI:
  + The UI uses both the Bitmap Display for graphics, and the Keyboard and Display MMIO Simulator for input/output. Console should output the same things as the Display Simulator, but it’s only there for debugging.
  + Entering any numbers must be followed by “Enter” to confirm the entry.
  + Canceling an entry can be done by entering something other than a number and pressing “Enter”.
  + The Bitmap Display will clear itself
  + Keyboard and Display MMIO Simulator will **not** clear itself.
* Playing the game:
  + Difficulty levels are ordered from easiest to most difficult (3 being the hardest).
  + Custom mode entries are either self explanatory, or are explained below.
  + The controls are shown to the user with the Display Simulator, but for redundancy:  
    W/w = move up  
    S/s = move down (capitals are for moving fast)  
    Anything else = stop moving
* Goal:
  + The goal of the game is to get the ball behind the enemy, while protecting your own goal
* Ball/Player Mechanics:
  + The ball moves at a set speed(initial delay), and every time the ball hits a wall (top and bottom of the play area), it will speed up, until it reaches its max speed (ie min delay).
    - This will return to the initial speed once a goal has been made.
  + When the ball collides with the player, the ball will bounce away from the player, and one of 3 outcomes can happen:
    - The ball hits the top or bottom: the ball will have 2 vertical velocity
    - The ball hits right in the middle: the ball will have 0 vertical velocity
    - The ball hits anywhere else on the front: the ball will have 1 velocity, and will vertically bounce away from the paddle (so if it hits the top it will travel upwards)
  + The player can either move slow or fast, depending on if they’re holding Shift when they input a movement
    - The player’s speed has no effect on the ball’s speed.
    - Note: If the player is holding down a movement key, and presses Shift, they must let go of that movement key and press it again else the movement wont register
      * This is an issue with the keyboard interface, not the code
    - Note2: The controls are the only thing about this project that are clunky, but I couldn’t figure out a better way of implementing them.
    - With a controller, the user could bind the joystick so that half tilts =w/s, and full tilts=W/S, or bind fast to one joystick and slow to the other. Steam’s Controller Interface should make this possible.
* AI/CPU player (player 2):
  + Ai level:
    - The AI level is a number from 0-20 that dictates how difficult the user’s opponent is. It works on a “movement opportunity” idea, where it will roll a number from 0-19, and if that number is <= the ai level, it will move faster.
      * It calculates this every time its able to move
  + AI chance not to move:
    - Self explanatory, it works the same as the ai level, but if it rolls, it wont move for that movement call.
    - This gets jittery at values above ~30-40
  + How it works:
    - Simply put, player 2 will always try and move where the ball currently is
      * Its very rudimentary, but so was the original Pong

# Writeup for project details

Note: I lightly documented things as I made the project, but if I were to go into great detail here, this part would be extremely long, so I will keep things in a mostly summary form. Obviously, some parts of the code could be better, but regardless I put way too much time into this project. I tried to make everything as modular as possible while designing this project

Architectural description

* Part 1 explains everything on the front end, but to go more in depth:
* How things are drawn:
  + Most of the foundations are ripped from the Simon Says project.
  + The DrawLine procedure was modified to allow dotted lines, with a gap between dots being able to be specified (internally the procedure is not DrawDottedLine.
    - The dotted lines with multiple dots per segment are drawn using the above with two for loops in DrawGraphicsInitial.
  + Numbers are drawn using a somewhat complicated system. First, the procedure DrawScore takes in an integer from PlayerScored, converts the int to a series of chars in a buffer, and calls OutText to draw the numbers.
    - The procedure is set up to only draw the numbers it needs to (so it will stop if it has to draw a digit other than a 0)
  + The players are initially drawn in DrawGraphicsInitial, and they are moved by drawing horizontal lines above and below them, with the color being applied to the line in the direction the player is moving, and the background color on the other
  + The ball is drawn in a similar way, except it will store the color it is currently occupying to refill it
  + Colors are grabbed from a color table, so certain objects can be changed easily
* Game Logic:
  + There are a couple of “stages” the game can be in
  + Initial:
    - This is pre-game, where the user can input parameters.
    - A Keyboard buffer and interrupts are used here, but aren't used for the actual game (due to things breaking)
    - After the user enters all parameters, the initial graphics are drawn
  + Pre-Round:
    - The player and AI can freely move for ~2 seconds until the ball is served.
    - Parameters that need to be reset or set a certain way are done so here (such as the ball delay, the ball y pos being set to the center for the AI, etc)
    - After those 2 seconds, the ball is served
  + Pong:
    - The game itself. There are 4 blocks of code that get called as much as the machine can call them. Each block is on a time delay of when it will actually execute, the machine is mostly checking if it can execute them yet
    - Block 1: Player 1 movement
      * This block handles the movement of player 1 (internally player 0). It looks at what the most recent key that was pressed was and passes it into Player0Handling, which moves the paddle, and returns if it should be a fast or slow movement.
      * There is a fast and a slow movement, which are defined by the delay the block is given, being 8ms for fast and 15ms for slow.
    - Block 2: AI / Player 2 movement
      * Similar to Block 1, except the delays are 8ms and 18ms.
      * Calls Player1AI, which handles moving P2 towards the ball.
        + This procedure also handles the AI movement calculations (explained in the End user writeup). It will move the player accordingly (or not at all if it fails the movement chance), and return if the movement should be fast or slow based on the AI level.
    - Block 3: Ball movement
      * Works similar to the above 2 blocks, calling BallMovement. This procedure checks for: top/bottom collisions, player collisions, and goals. It will bounce the ball appropriately. It also moves the ball as well.
      * BallSpeedDampener is also called in this, and it handles sending either a 1 or 0 to reduce the speed. It pulls these numbers from a preset series of bytes. Since the speed is controlled by a delay, reducing it by 1ms every cycle would accelerate the ball, which is not good.
        + The speed is reduced every time the ball hits either the top or bottom of the screen, but of course back end sometimes it isn't reduced when BallSpeedDampener returns a 0.
      * BallMovement was by far the hardest procedure to make and get right, as every case that could happen had to be handled by it
    - Block 4: Ball collision checking
      * This block simply does what block 3 does, calling BallMovement, but is checked every ~~4ms~~ 7ms, and only checks when the ball’s x is at the player (1 unit in front to the edge of the screen).
      * This block passes in a bool to BallMovement to only collision checking.
        + In this state, the ball will not move unless it is going to collide with a player, which in this case it will do the same thing Block 3 would
      * This block does not speed the ball up
  + Somebody Won:
    - Self explanatory. This is checked for at the start of Pre-Round
    - Also handles if the user would like to continue or quit
* Ball stuff:
  + The ball has an x and y velocity, and where it will next move is calculated using those numbers. Those numbers are used for collision checking (as well as the originally x and y coordinates for some cases), and if a collision happens the appropriate velocities / logics are applied and the x and y positions are recalculated. First the ceiling and floor are checked, then the paddles are checked. And if there is no paddle collision then its checked if there was a goal.
  + The speed is increased each time the ball hits the ceiling or floor
  + Collisions with paddles are checked for every ~~4ms~~ 7ms

Test/verification/build process

* Any procedures that could either be tested independently, or didn't need to be removed from the program to be tested include:
  + BallSpeedDampener, Player1Ai, UpdatePaddlePostions, DrawBall, PlayerScored, StartRound, UserInputParameters, DrawGraphicsInitial, DrawScore, DrawBox, DrawDottedLine, DrawDot, StoreColorAtCoord, GetColor, OutText, GetNumsFromKeyboardSim, PrintToKeyBoardMMIO, pause, and clear\_buffer.
  + Testing would have gone as far as just making sure that with different inputs, it gave the right outputs / did what was expected.
  + This isn’t to say that it didn’t take a long time to test these, it's just that they were a lot easier/simpler to test
* Any other procedures were vigorously tested in numerous ways.
  + Includes: Player0Handling, BallMovement, technically main
  + Some procedures still have commented lines that were used for testing. Obviously these test lines shouldnt be used by the end user (for example, some of them use $k1 because it's not touched anywhere else).
  + An example, the BallMovement procedure was probably the hardest to test out of any procedures. It handles both the collision checking and movement of the ball. Every issue you could think of happening with it probably happened at some point. I tried to make it as well put together as possible, but that procedure is easily the messiest.
    - Very late into making this, I made it so it could only check for collisions with players and act accordingly (refer to Block 4 above). This only took me ~30 minutes to implement because of how well the procedure was put together. I did this to fix a bug that was happening (explained in the debug section)
* Note: Even though this project took me forever to do, having a lot of procedures really helped in the long run with testing and staying organized (hence there only being 3 procedures i vigorously needed to test)
* Procedures that I wrote from scratch:
  + Player0Handling: handles the player movement. Moves player 0’s up or down depending on what key was last pressed. Also handles the logic of where the paddle can move
  + BallMovement: procedure handles ball collisions and movements.
    - Checks for collisions with the top and bottom, player paddles, and for goals.
    - Uses DrawBall to draw the ball, as well as to store the color it wrote over to redraw the pixel on the next call.
    - Can be set to only check for paddle collisions, and to move the ball only in such case, but not move the ball otherwise
    - Outputs if there was a score & who scored, as well as if the ball should speed up (if the ball hit the top or bottom).
    - Handles the 3 paddle collision types, which is explained under “Ball/Player Mechanics”
  + Main: mostly explained above in Game logic. It's the driver/controller procedure.
  + BallSpeedDampener: procedure takes a series of bytes from .data, and returns each bit in order, one bit on each call. This is to dampen the speed of the ball (explained above)
  + Player1Ai: like Player0Handling, but less complicated. It simply moves player 1 towards the ball’s current y coordinate
  + UpdatePaddlePostions: procedure handles moving the paddles by drawing 2 horizontal lines, one above and one below. One line is background colored, and the other is paddle colored. Also handles the logic of if the paddle can move
  + DrawBall: Draws the ball.
    - Calls DrawDot.
    - Also calls StoreColorAtCoord to get the current color, so the ball can fill in the tile that it is currently occupying on the next call of BallMovement
  + PlayerScored: called from main when a player scores.
    - Handles incrementing the counter and calling DrawScore to draw the new score
  + StartRound: Serves the ball. Randomly picks a valid y coordinate and y velocity to serve at.
  + UserInputParameters: procedure for entering the custom parameters.
    - User can pick from a pre-set difficulty, or a custom one.
    - Makes sure custom parameters are valid
  + DrawGraphicsInitial: draws the initial graphics, including: numbers, lines, paddles.
    - Uses a double for loop with DrawLineHorizontal to draw the lines with multiple dots per gap.
  + DrawScore: Draws a score. Only redraws numbers it needs to.
    - Calls OutText to do this. Procedure also handles turning an integer input into a list of ascii numbers
  + StoreColorAtCoord: potentially misleading name, procedure gets the color at the given x,y coordinate, and stores it to “ballLastColor” in .data
  + GetNumsFromKeyboardSim: procedure to get numbers from the keyboard simulator.
    - Takes up to 10 digits. Allows the user to re-enter a number if it was invalid
  + PrintToKeyBoardMMIO: prints a string to the keyboard simulator
* Procedures that were reused from past projects:
  + DrawBox: self explanatory
  + DrawDottedLine (modified): self explanatory, modified to be able to draw dotted lines
  + DrawDot (slightly modified, combined it with the procedure to calculate the pixel’s coordinates): self explanatory
  + GetColor: returns a color from a table in .data dependent on an integer input
  + Pause: self explanatory
  + Clear\_buffer: clears the keyboard buffer. Only used for pre-round inputs
* Procedures ripped from other materials:
  + OutText: taken from the last part of simon says, but it has been modified somewhat
  + The exception handler: i added the code to handle keyboard inputs (which is only an interruption pre-round), but everything else was taken from exceptions.s. Technically most of it isn’t even necessary, but I figured I might as well leave it

Debug/issues:

* Issues:
  + Mars/MIPS not having a way to read key states (key press, key release).
    - Maybe I’m just blind, as I couldn't find a way to implement this, but this not existing made the controls feel clunky.
  + Ball Acceleration:
    - By subtracting 1 from the delay each time, it made the ball speed up way too much. I made the procedure BallSpeedDampener to counteract this. Simply put, it reads from a series of bytes, and gets each bit in order (1 bit from each call), to determine if it should subtract 1 or 0 from the delay. I originally wanted to make this a logarithmic function, but I quickly realized it wasn't worth doing. The last 4 bits are then repeated until the current delay is <= the min delay. Also, in UserInputParameters, it sets it up so it wont start at the start of the bytes for every delay. Different starting delays will still have different results.
      * So for example, current delay=10 for a starting delay of 20 vs 30 will be different. The higher the starting delay, the sooner it reaches its minimum deceleration (0001).
* There were way too many bugs to cover them all, but here are a few notable ones:
* Bugs with ball eating pixels
  + Obviously, initially the ball would eat pretty much any pixel, or it wouldn't collide with the wall right, or anything you could think of.
  + The more notable bugs of this were:
    - When the ball would collide with the wall, then the paddle, in a single BallMovement call, sometimes it could eat the paddle’s pixel
    - Another was from the ball not colliding with the paddle when it physically was (since calculations originally only looked at where the ball would be)
      * This was fixed by having a separate “if” area that would handle if the ball’s calculated coordinates were not in the paddle, but its current coordinates were, which it would fill the pixel with the paddle color after moving instead of the stored color (which was probably background)
    - Another was because sometimes the player could move faster than the ball. So I implemented Block 4 to fix this
* Keyboard interrupt issue
  + While keyboard interrupts were being used, if the player spammed a key or sent too many inputs too quickly, the program would just break. Something went wrong either with the stack or the stack pointer, as $ra would sometimes just be 1 or 0
  + Obviously, I fixed this by just reading the key when I needed to, since there wasn't a need for a buffer while playing the game. However, I had to scrap 2 player mode because of this.
* Bugs that still exist
  + The ball can eat part of the border if it hits the 2nd pixel of a paddle from the wall while the paddle is up against the wall. This is very rare, and it's nearly impossible to notice since the ball replaces the pixel on its next movement call. So only at slow speeds is it noticeable
  + The ball might still be able to eat a pixel on the farthest back column of the players under specific circumstances. I’m pretty sure I fixed it though
  + I’ve never encountered this outside of manually setting coordinates, but if the ball somehow got outside the play area stuff could easily break
    - This could happen if a y velocity >2 was implemented
  + Not a bug, but the score will overflow after 5 digits. Internally it's still stored and used correctly, but the display will overflow to 00000, and count normally from there.
  + Right before submitting, the audio was bugging out a little, so I updated the ball collision check to check every 7ms instead of 4ms.

# Conclusion

To keep it brief, I really enjoyed working on this project. Infact, I think I enjoyed it *too* much, considering I probably spent anywhere from 60-100 hours working on it in total. I really didn’t want to cut corners, so I ended up making a project probably way bigger than it needed to be. Regardless, I’m satisfied with the final product.