Tetris in Assembly

Design Brief

Memory Layout

The `.data` segment for static data, we store hardware constants —display base address `ADDR_DSPL`, keyboard address `ADDR_KBRD`, two 8-byte buffers for the `current_piece` and `rotated_piece`, and three collision masks `grid_below`, `grid_left` and `grid_right`. Additional space stores the *GAME OVER* bitmaps and musical note constants defined via `.eqv` commands. Lastly, throughout the program, key pointers and counters are stored in

> **1: Callee-saved Registers**
>
> 1. `$s0` points to the `current_piece` buffer
> 2. `$s1` stores the color of `current_piece`
> 3. `$s2` points to the `rotated_piece` buffer
> 4. `$s3` points to the collision masks `grid_below`, `grid_left` and `grid_right` depending on the direction we are checking for collisions in
> 5. `$s4` stores the number of lines cleared and is used to calculate the adaptive gravity speed (i.e increase it)
> 6. `$s5` stores the gravity counter and is used to determine when to apply gravity
> 7. `$s6` points to the keyboard
> 8. `$s7` points to the display

Main Control Flow

The `main` routine initializes the display and keyboard, and points `$s3` at `grid_below` since that is the direction the piece will move in initially. `main` then draws the walls and checkerboard pattern, plays the intro jingle, and resets the spawn coordinates ($(x, y) =$ `($a2,$a3)`) and the line + gravity counters. Finally, `main` picks a random piece and color, spawns it, and jumps to `game_loop`.

Inside `game_loop`, we check the keyboard for inputs. If no key is pressed, we wait 50ms (hence increasing `$s5`) before checking again. If `$s5` exceeds 1900ms, we call `gravity` to move the piece down automatically. When a key is pressed, we jump to `process_key` to handle left/right movement, soft/hard drops, rotations, restarts, and quits.

During movement, we erase the current piece, populate a 4×4 mask of the direction we want to move in, test for overlaps, and apply or reject the movement. When a piece reaches the bottom row or lands on another piece, the piece is locked in place and the locking sound effect is played. If there are any completed rows, they are cleared and the row clearing sound is played. The spawn coordinates are reset and a new piece is spawned at the top.

The game ends when a piece is unable to spawn or a player quits via `q`, navigating to the game over screen and plays the outro song. During this time, the player can press `r` to restart the game after the outro ends. During the game, the player halso as the option to restart the game via `r` if they make a mis-input.

Tricks and Optimizations

2: Tricks and Optimizations

1. **Bitmask Representation**: Each tetromino is four half-words (rows) with bits 3–0 (due to Little Endian) as occupied columns. Drawing and collision tests are done by shifting a 1-bit mask and `AND`'ing against these rows. This avoids nested loops and per-cell branches
2. **Fast Math**: since everything is in powers of 2, we use left and right shifts (`sll` and `srl` for multiplication and division, because these operations much more optimized for our architecture
3. **Optimized Routines**: for collision detection, we use fast bitwise operations (`AND` and `OR`), and the piece rotation routine directly hardcodes the transformation $(i, j) \to (3 - j, i)$ using `AND, SRL, SLL, OR` operations on registers —no loops are needed

Sound Effects Used

3: SFX

- **Intro**: Supercell
- **Piece Lock**: Mac Startup chime
- **Row Cleared**: SMB Mushroom SFX
- **Game Restarting**: Windows XP Shutdown sound
- **Game Over Outro**: *Die Forelle* (Samsung washing machine song)