Considerations:

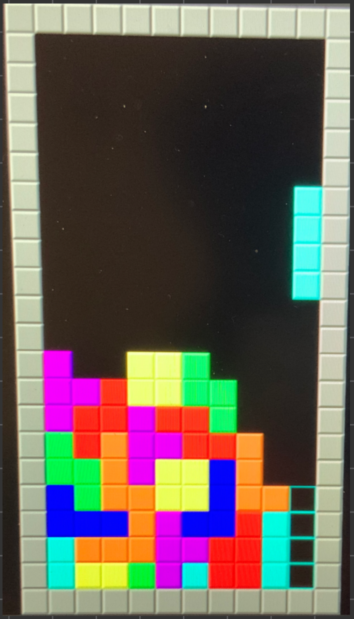
-each piece has diff # of rotation states   manually store each rot. state in ROM. everytime you press button it'll "rotate"

Rotation clipping: if the rotate state (one we decided earlier) clips outside, don't allow rotations

Falling and permanent system:

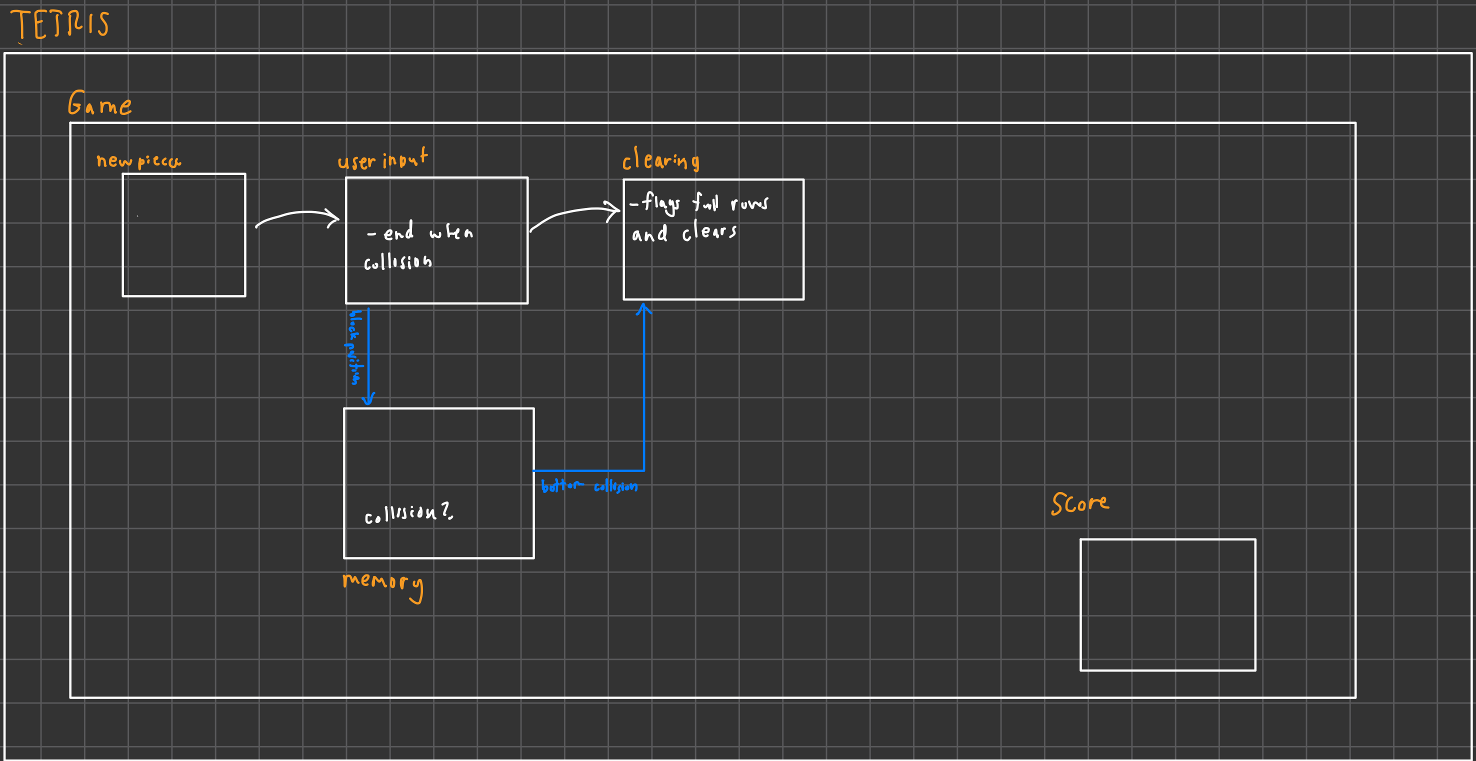
Clearing lines: detect if any line is full, clear it, shift down, repeat 4x ☆

Left/right + down at same time will cause clipping



10x20

Different Modules:



new piece :

- runs a fast clock and constantly gives output. -need new piece next piece system.
  - runs a loop that resets if @ 7.

User input :

Tetris Game.sv : Top level, game state FSM (pause, reset, etc)

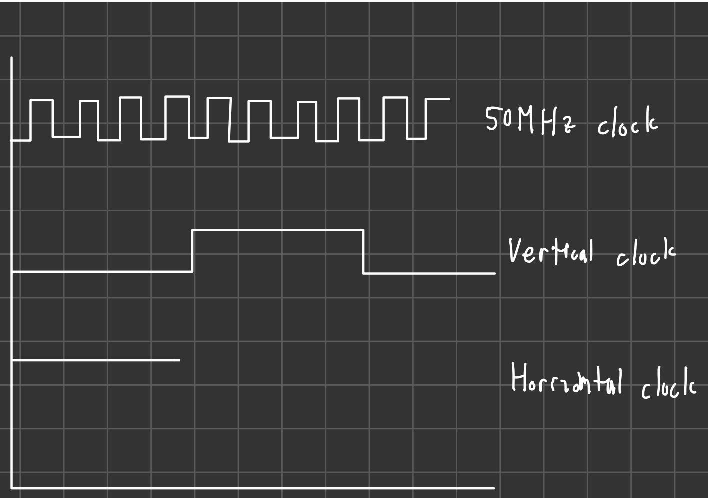
Game.sv : Core game logic (piece motion), line clears, input)

- Piece rom : ROM of 7 pieces x4 rotations
- Piece logic : Converts ROM to actual (x,y) block coords
- Board.sv : Stores 10x20 playfield

Score.sv : score (inputs are from game line clearing, outputs score)

vga_ctrl :

tick_gen.sv : vertical and horizontal clocks to avoid diagonal glitching



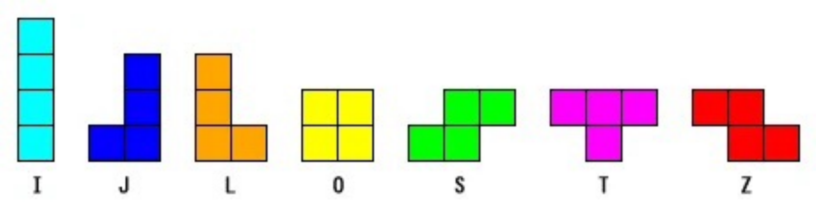
- vert tick happens every 750ms ← 15000 ticks
- horizontal tick happens every 20ms ← 400 ticks

- if both are gonna be on at the same time, vertical takes priority
  - this can be done in game.sv with an "if" else block

- don't allow holding for left/right. game ensures that it gets a "low" press before allowing to move again

Piece_rom: Used to lookup and spawn pieces.
- has each pieces ID, rotation, colour, and matrix.
- input a piece ID, rotatinon, and row_index, can get out the data for that row, and colour.

Ex. input ID 000, rotation 00, row-index 00, output BLUE, 0010



000 001 010 011 100 101 110

-All rotations will be CCW

```
00  XXXX
01  XXXX
10  XXXX
11  XXXX
```

Piece 000:

```
0010    0000    0010    0000
0010    0000    0010    0000
0010    1111    0010    1111
0010    0000    0010    0000
```

Rotation 00   Rotation 01   Rotation 10   Rotation 11

- Defined piece_row as [0, 3] so that when reading it from rom, we "see" each 4x4 as

0,0	1,0	2,0	3,0
	...		
			3,3

- mimics the board.

Piece 001:

```
0000    0000    0000    0000
0100    0010    0000    0011
0111    0010    0111    0010
0000    0110    0001    0010
```

Piece 010:

```
0000    0000    0000    0000
0001    0110    0000    0010
0111    0010    0111    0010
0000    0010    0100    0011
```

Piece 011:

```
0000
0000
0110
0110
```

Piece 100:

```
0000    0000
0000    0010
0011    0011
0110    0001
00, 10  01, 11
```

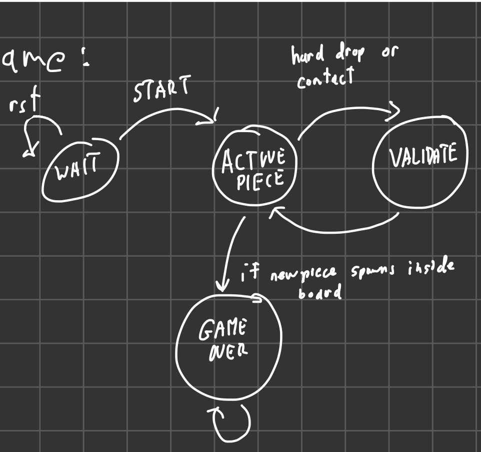
Piece 101:

```
0000    0000    0000    0000
0000    0010    0010    0010
0111    0011    0111    0110
0010    0010    0000    0010
```

Piece 110:

```
0000    0000
0000    0001
0110    0011
0011    0010
00, 10  01, 11
```

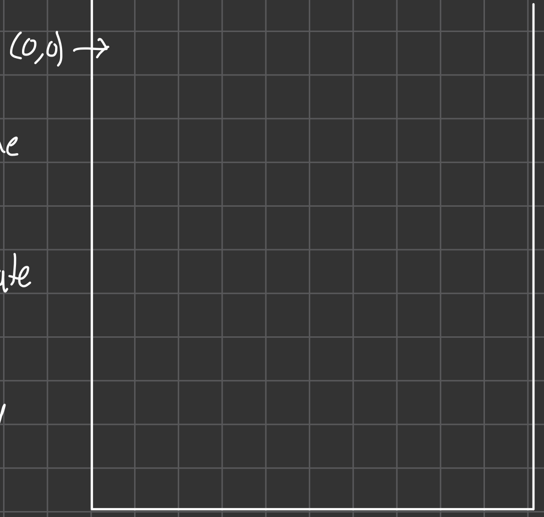
game:



board.sv:

- defines 2D array of memory which starts blank.
- combinational reads: given an x,y, coordinate, simply returns the colour and whether cell is occupied or not.

- after rst, wait in a wait state until game.sv goes into valodate state.



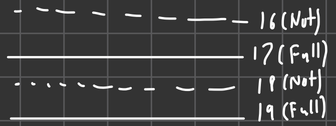
WRITE: need to write current pieces colour into 2D array at each cell.

- write each (x_n, y_n) into memory from bottom up.

CLEAR LINE:

- scan from bottom row up.
- if row is not full, copy it to lowest available row
- if row full, skip it (don't copy it)

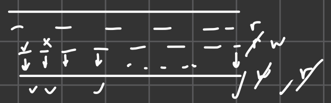
Count how many lines were cleared.
Write



write_row @ 19:
Iterate read-row from 19 down to 0
For each read_row, if it's full, skip
if not full, copy it to write_row then decrement write_row

x <= x + 1
if (board [read_row] [x]

if [ ] occapied, scan next.
scan until we know entire row full, then move read_row up.



- Unoccupied → occupied → copy
- Unoccupied → empty → stop
- Occapied → empty → copy
- Occapied → full → skip.

- entire row empty? → WAIT
- entire row full? → read_row --
- neither? COPYLINE

FETCH:

- use "current_piece" value to look-up piece data.
- store all 4 blocks in current_x[i] and current_y[i] for current piece.
- whenever we scan 1 of the pieces, also check that it didn't spawn in an occupied spot. If so, game over

MOVE

- fast clock continuously cycling through a occupied flag checker for all 4 pieces? NO!

- instead, check for collisions only once after every vertical tick.
  ↳ Add 4 more inputs to board, so that this check can be done "instantly" after each vert_tick

```
-if no vertical movement, check horizontal movement.
-if (horizontal_flag && movement_valid)
    if (left && left_allowed)
      ______
    if (right && right_allowed)
      ______
  else
    if (rotation &&rotation_valid)
      -reassign new current_x[n] current_y[n] with ROM.
```

Rotation:

- relative "anchor" of top left grid in game module that updates every time block moves.
- when rotation is called, we add 1 to rotation, then go into a rotate_wait state.
- in rotate_wait, we read from ROM (the current piece but rotated), and add every value to our relative anchor for a "future_x_n, future_y_n".
- board module will let us know if rotation allowed then raise a flag saying it's done.
- when done game FSM will continue, either going back to AP_MOVE_NOT_ALLOWED, or a new rotation state where we assign the new current values.
- if rotation wasn't allowed, subtract 1 from rotate!

Fast drop:

- if pressed go into fast drop state
- while crash is 0, keep decrementing future y_n by 1.
- every time after fastdrop pressed, set fastdrop allowed = 0, and only set back to 1 if button released.

ACTIVE PIECE:

- pass current piece to piece_rom and spawn it @ set location (top middle)
- if new piece spawns in occupied matrix, game over.

- constantly check horizontal flag and vertical flag to detect block drop by 1 unit or if player wants to move piece left/right. Priority system of vertical flag over horizontal
- check for hard drop key OR if vertical tick is high and spots below active piece are occupied. If so, go to next state

VALIDATE:

- update board.sv matrix w/ new occupied spots and check if there are any full rows, working from bottom up. If so, check the next 3 rows up and shift entire matrix down.
- output how many lines were cleared and output for score
- return to active piece

CONSIDERATIONS

ACTIVE PIECE.

- if hard-drop, check all rows below it, and only update y after deciding where it should land,

VALIDATE

- make sure active_piece flag is cleared when coming into this state so new piece doesn't spawn.

TETRIS - top level module

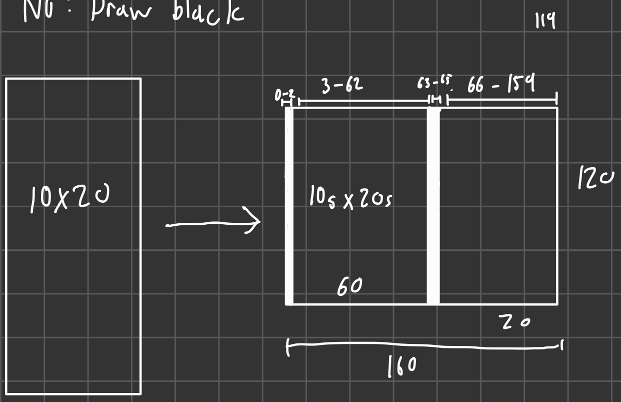
- instantiate game.sv and vga_adapter

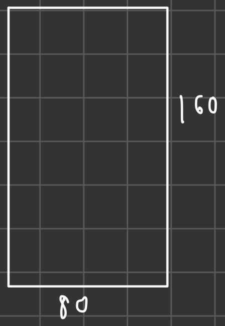
VGA

- vga adapter blindly follows instructions
- it simply scans every pixel starting from (0,0) to (max_x, max_y) (or whatever the playable tetris dimensions are) and asks:
    - is this pixel on the locked board or current_x, current_y?

      YES: Draw colour.

      NO: Draw black





- each board.sv entry translates to a 6x6 area on screen.

EX. (0,0) translates to a 6x6 grid w/ top left @ (3,0) and bottom right at (8,5)

$$\text{board\_x} = (\text{screen\_x} - 3) \div 6$$
$$\text{board\_y} = \text{screen\_y} \div 6$$

- if during a validate state vga screen starts spazzing, introduce a rdy/en signal output from game.sv state allowing for SCANDRAW to happen.