A Tetris Game in Assembly Language

Learning Goal: Write a complete program in assembly language and run it on a NIOS II processor.

Requirements: nios2sim Simulator (Java 10), Gecko4Education-EPFL, multicycle Nios II processor.

1 Introduction

In this lab, the goal is to implement a simplified version of the **TETRIS** game in assembly language. At the end of the lab, you should be able to play it on **Gecko4EPFL** board.

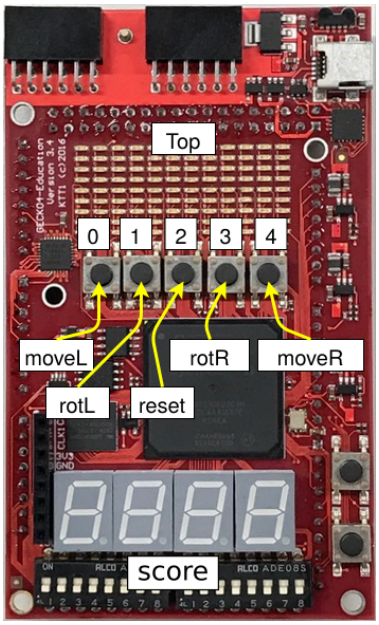


Figure 1: Gecko4EPFL layout and the role of interfaces.

1.1 General Description

The goal in **TETRIS**[1] game is to arrange **tetrominoes** (this is how the game pieces in **TETRIS** are called) to form continuous lines that span all the way from the left to the right edge of the game screen. Once one such line is formed, it will disappear and the game score will be incremented. During the game, the tetrominoes can be moved to the left or to the right (horizontally), or rotated, or simply let fall down. Once the game screen becomes full and the new tetromino is no longer able to move down from the top of the screen, the game ends. The longest one plays, the higher the score and thus better the outcome.

Figure 1 shows the Gecko4EPFL board and the role of the buttons. The game will be displayed on the LED array of the Gecko4EPFL. Initially, the screen will be empty, the score will be zero, and the first tetromino will be generated close to the top of the screen. Each LED represents one pixel, and each tetromino, as will be explained later, is made of 4 pixels. To make the game implementation simpler, you will use only the top 8 rows of the LED screen (96 LEDs in total). From the initial tetromino location, you can move or rotate the tetromino, or let it fall until it reaches its final position. Then, the game will generate a new random tetromino. Once you line up the tetrominoes to form a full line, the line will be removed, and all tetrominoes stacked on the top of it will go down by one pixel precisely. One line disappearing will result in the score being incremented (increased by 1 point). The game continues until a newly created random tetromino can't fit the game screen.

You also have at your disposal a NiosII simulator, which has consistent behavior with the Gecko4EPFL and should help you debug your program. You can download the simulator from the moodle page of the course.

In the following sections, we will present the precise rules for the game. You must follow them in your implementation of the game.

[1] https://tetris.fandom.com/wiki/Tetris_Guideline

1.2 Constants

To improve the readability of your code, you can associate symbols to values with the `.equ` statement. The `.equ` statement takes a symbol and a value as arguments. For example, the line below

```
.equ SCORE,  0x1010
```

will associate value `0x1010` to symbol `SCORE`. In the code, whenever you write `SCORE`, that will have the same effect as if you write `0x1010`, but your code will be much more readable and easier to update. Example of use:

```
    ldw  t1,   SCORE (zero)    ; load the score in t1
```

We have prepared for you a list of useful symbol/value pairs. **For correct grading of the game, we strongly advise you to use the list below, without any modification! If you choose different symbol names or values, the grader may fail and you may lose points.**

```
;;    game state memory location
.equ T_X,           0x1000    ; falling tetromino position on x
.equ T_Y,           0x1004    ; falling tetromino position on y
.equ T_type,        0x1008    ; falling tetromino type
.equ T_orientation, 0x100C    ; falling tetromino orientation
.equ SCORE,         0x1010    ; score
.equ GSA,           0x1014    ; Game State Array starting address
.equ SEVEN_SEGS,    0x1198    ; 7-segment display addresses
.equ LEDS,          0x2000    ; LED address
.equ RANDOM_NUM,    0x2010    ; Random number generator address
.equ BUTTONS,       0x2030    ; Buttons addresses

;;  tetromino type enumeration
.equ C, 0x00                  ; carre (square)
.equ B, 0x01                  ; bar-shape
.equ T, 0x02                  ; t-shape
.equ S, 0x03                  ; s-shape
.equ L, 0x04                  ; l-shape

;; GSA type
.equ NOTHING, 0x00            ; the array location is not occupied
.equ PLACED,  0x01            ; occupied by a nonmoving object
.equ FALLING, 0x02            ; occupied by a falling object

;; orientation enumeration
.equ N,  0x00                 ; north
.equ E,  0x01                 ; east
.equ So, 0x02                 ; south
.equ W,  0x03                 ; west

;; Rotation enumeration
.equ CLOCKWISE, 0             ; clockwise direction of rotation
.equ COUNTERCLOCKWISE, 1      ; counterclockwise direction of rotation

;; Actions over tetrominos
.equ moveL, 0x01              ; move left, horizontally
.equ rotL,  0x02              ; rotate counterclockwise
.equ reset, 0x04              ; reset the game
.equ rotR,  0x08              ; rotate clockwise
```

```
.equ moveR, 0x10              ; move right, horizontally
.equ moveD, 0x20              ; move down, vertically

;; Collision return ENUM
.equ W_COL,   0x00            ; collision on the west side of tetrominoe
.equ E_COL,   0x01            ; collision on the east side of tetrominoe
.equ So_COL,  0x02            ; collision on the south side of tetrominoe
.equ OVERLAP, 0x03            ; tetromino overlaps with something
.equ NONE,    0x04            ; tetromino does not collide with anything

;; start location
.equ START_X, 6               ; start tetromino x-axis coordinate
.equ START_Y, 1               ; start tetromino y-axis coordinate

;; standard limits
.equ X_LIMIT, 12
.equ Y_LIMIT, 8

;; game rate of tetrominoes falling down (in terms of game loop iteration)
.equ RATE, 5
```

1.3 Formatting Rules

In the rest of the assignment, you will be asked to write several procedures in assembly language. If you implement them all correctly, you will be able to play the game using your Gecko4EPFL board. **To enable correct automatic grading of your code, you must follow all the instructions below:**

- surround every procedure with BEGIN and END commented lines as follows:

```
; BEGIN:procedure_name
procedure_name:
  ; your implementation code
  ret
; END:procedure_name
```

  Of course, replace the `procedure_name` with the correct name. **Please pay attention to spelling and spacing of the opening and closing macros.**

- If your procedure makes calls to other, auxiliary procedures, all those auxiliary procedures must also be entirely enclosed between begin:helper and end:helper comments. The auxiliary procedures may have whatever name you choose. You can also add any newly defined constants in the helper block.

```
; BEGIN:helper
my_helper_procedure_name1:
; your implementation code
ret

my_helper_procedure_name2:
; your implementation code
ret
; END:helper

; BEGIN:procedure_name:
```

```
procedure_name:
; your implementation code
call my_helper_procedure_name1
; your implementation code
ret
; END:procedure_name
```

- Have all the procedures inside a **single** `.asm` file.

Our grading system will check each procedure individually and separately from the rest of your assembly code.

2 Rules

2.1 Terminology

A tetromino is a game piece in the **TETRIS** game. It is called tetromino because it is composed of four small squares. In our game, we will have five different tetrominoes, shown in Figure 2.

A falling tetromino is the name given to the tetromino currently falling, which means that it does not touch anything yet (or we don't know it yet) and will periodically be brought one pixel down. It is the only tetromino not yet placed; it will be shaded in dark blue in the figures.

A placed tetromino is a tetromino that has touched either the bottom of the game screen arriving from above or another tetromino arriving from above. A placed tetromino might end up cropped if it formed a full line that got removed from the game screen.

An anchor point is the reference point for a tetromino; we will mark it in purple in the figures. We draw tetrominoes with respect to the location of their anchor points. Additionally, we rotate tetrominoes around their anchor points.

A full horizontal line is a line such that on the game state array (GSA, see Section 2.4 for details) we have

$$GSA(x, y) = 1 \; \forall x \in \{0, 1, ..., 11\} \text{ for some y} \in \{0, 1, ..., 7\}.$$

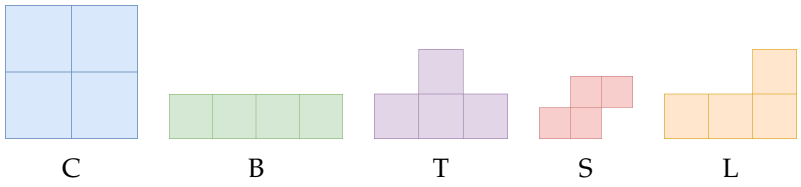


Figure 2: Tetrominoes.

2.2 Tetrominoes

The tetrominoes we use in this version of **TETRIS** are shown in Figure 2. They are square-shaped (C), bar-shaped (B), T-shaped (T), S-shaped (S), and L-shaped (L). Table 1 shows how to encode every tetromino using symbol/value pairs. Tetrominoes can be oriented towards north (N), east (E), south (So), or west (W). The tetrominoes in Figure 2 are **all oriented towards north**. Table 2 shows how to encode symbol/value pair per direction.

value	tetromino shape (type)
0x00	C
0x01	B
0x02	T
0x03	S
0x04	L

Table 1: Symbol/value pairs to define all tetromino shapes.

Besides the type and orientation, every tetromino has something we call an **anchor point**. This point is uniquely defined for every type of tetromino. See Figure 3(a) for the location of anchor points in tetrominoes oriented towards north.

value	orientation
0x00	N
0x01	E
0x02	So
0x03	W

Table 2: Symbol/value pairs to define all tetromino orientations.

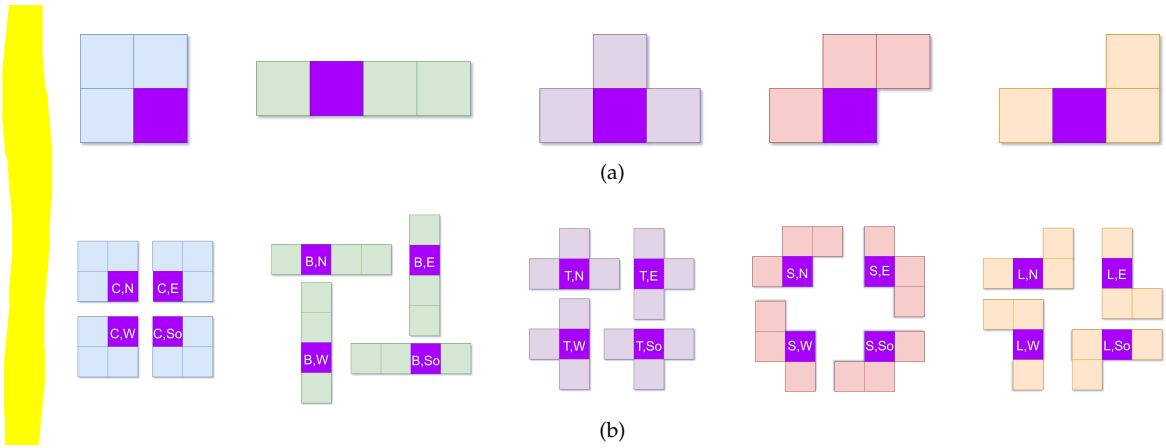


Figure 3: Tetrominoes and their anchor points and orientations. (a) Orientation to the north. (b) All orientations.

Whenever a new tetromino is generated, its shape will be randomly selected from {C, B, T, S, L}. However, regardless of the shape, it will always be oriented towards north and its anchor point will be at **(6,1)**. This initial anchor point location is marked as `START` in Figure 4.

All possible orientations for all possible tetrominoes are shown in Figure 3(b), please use exactly those ones as using a different but valid set of orientations will not be accepted by the grading system. You can see more example of different orientations in the appendix A.

The type, orientation, and the location of the anchor point fully define one tetromino.

2.3 Coordinate system

The coordinate system for the game is shown in Figure 4. The top left corner of the LED array is the coordinate system origin (0,0), the x-axis grows rightwards and the y-axis grows downwards. On Gecko4EPFL, tetrominoes start falling from the top (Figure 1 shows where top is).

2.4 Game State Array

The Game State Array (GSA) contains all the information to describe what is going on the screen. It has one element per screen pixel; in total 96 elements of 32 bits. The relation between its elements and the LEDs pixels can be found in Figure 5. For example, the element in location (1,4) is at location 12 in the GSA and in LEDS[0][12], counting from 0. Similarly, the element 51 in the GSA, so at location (6,3), corresponds to LEDS[1][19]. Each element in the GSA can be **NOTHING**, **PLACED** or **FALLING**, as summarized in Table 3. Only one tetromino can be FALLING at a time, while all the other tetrominoes are PLACED. Thanks to this representation, it will be easy afterwards to detect collisions, and to move the falling tetromino.

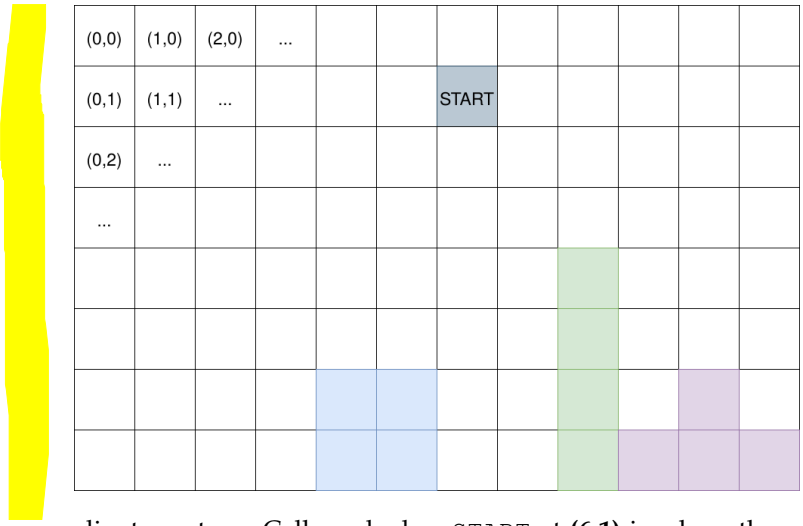


Figure 4: Game coordinate system. Cell marked as `START` at **(6,1)** is where the anchor point of every new tetromino should be.

value	GSA element
0x00	NOTHING
0x01	PLACED
0x02	FALLING

Table 3: GSA elements can take one of these three values.

Figure 5: Mapping between the GSA indices and the location on the LEDs.

2.5 Falling tetromino description in RAM

To describe a falling tetromino, we need to specify the following information:

- current x-coordinate of the anchor point, $0 \leq x \leq 11$
- current y-coordinate of the anchor point, $0 \leq y \leq 7$

- shape (type) $t \in$ {T, B, S, C, L}
- orientation $o \in$ {N, E, So, W}

This data will be located in RAM memory, as shown in Table 5. The addresses at which these informations are to be held are identified using symbols **T_X**, **T_Y**, **T_orientation** and **T_type** (See also the list of symbol/value pairs in Section 1.2).

2.6 Rotating a falling tetromino

In order to rotate the tetrominoes, we base the rotation system on the anchor points: during rotation, the anchor point remains fixed. We can only rotate a falling tetromino, and we can rotate it by 90° at a time, clockwise or counterclockwise. We change the orientation of the falling tetromino by changing the orientation attribute in the tetromino data structure in RAM memory. Examples of possible transitions are N $\rightarrow$ E, W $\rightarrow$ So, etc. Figures 6 illustrates rotation; other cases are similar. Please note that the location of anchor point does not change during rotation.

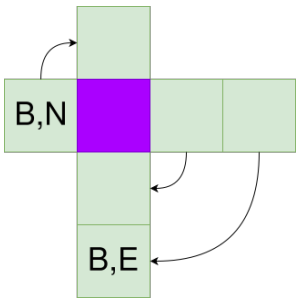


Figure 6
Bar, from N to E, clockwise.

Finally, when rotation occurs, we need to be aware of some edge cases. For example, see Figure 7: if you rotate the bar tetromino when it is vertical and close to the edge of the screen, it might end up outside the screen. In this case, the policy we adopt is to try to move it horizontally towards the center of the screen, pixel by pixel, by a maximum of **2** pixels. If, after trying to move it one pixel or at most two pixels, the tetromino fits entirely the screen again, all is good and the tetromino is rotated and drawn at its new location. However, if even after moving it for two pixels, the tetromino still does not fit because, for example, another tetromino is obstructing the way, then the rotation should not happen at all and the tetromino should be left at its initial position (the position before any movement or rotation).

2.7 Collision detection

value	collision
0x00	W_COL
0x01	E_COL
0x02	So_COL
0x03	OVERLAP
0x04	NONE

Table 4: Symbol/value pairs to represent various collision scenarios.

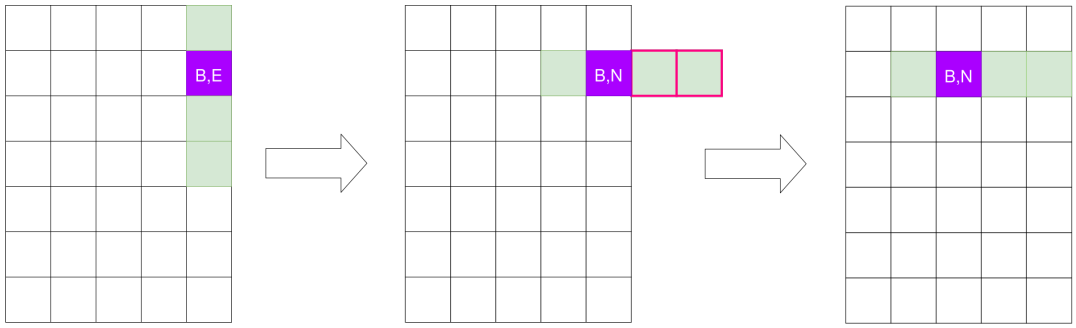


Figure 7: Tetromino rotation when things get tricky.
Left: Tetromino bar (B) on the right edge of the screen, oriented towards east. **Middle**: The tetromino is rotated counterclockwise (towards the north), and is therefore wrongly placed (partly out of the screen). This state will not be drawn on the LEDs. **Right**: the situation (error) is detected by the game logic and the tetromino is placed correctly (rotated, and moved towards the center of the screen).

Collision detection is the central aspect of the **`TETRIS`** game, as it prevents prohibited placement of tetrominoes and triggers the generation of a new tetromino. We can identify four different scenarios involving collisions, listed in Table 4:

- W_COL: at least one of the pixels on the west (left) side of the tetromino is either occupied or a wall. This would result in an overlap if one tries to move the tetromino towards the left, otherwise it has no effect.
- E_COL: at least one of the pixels on the east (right) side of the tetromino is either occupied or a wall. This would result in an overlap if one tries to move the tetromino towards the right, otherwise it has no effect.
- So_COL: at least one of the pixels on the south (bottom) side of the tetromino is occupied. If no action is taken, this will result in an overlap in the next falling step.
- OVERLAP: at least one of the pixels of the tetromino **already** overlaps with a placed pixel or is outside the screen.
- NONE: none of the above.

Figure 8 shows an example of E_COL and OVERLAP while Figure 9 shows an example of a So_COL.

Note that some collision scenarios, even when they occur, do not necessarily need to be signaled. As an example, if the falling tetromino has an E_COL (as in the **Left** side of Figure 8), but we are currently trying to move it towards the left, then this collision scenario should not be signaled. Therefore, we will only check for collisions relevant to the intended movement of the tetromino. A more detailed description that will be needed for the implementation of the collision detection is presented in Section 8. Finally, since north collisions never trigger any event in the game, we will not consider them.

2.8 Horizontal lines

When a horizontal line is completed, it needs to be detected and removed, by deleting the line pixels and moving down by a single pixel all the tetrominoes placed above the line. This is illustrated in Figure 10. In this figure, the blue tetromino is the falling tetrominoe. It is first moved down, and then, when we try to move it down a second time, it will result in an overlap with the ground, and it will generate a So_COL. Since it can't be moved down, it triggers the full line removal mechanism, which blinks the full

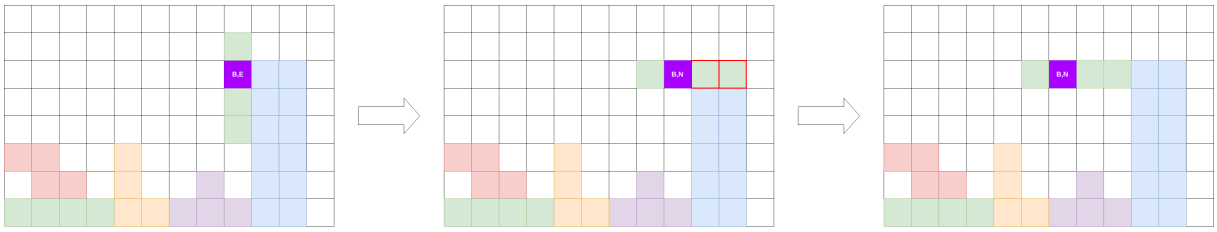


Figure 8: Examples of collision scenarios.
Left: The tetromino has an E_COL but, unless we have to move it to the right, it will not be detected. **Middle**: When the falling tetromino is rotated, an OVERLAP is detected. **Right**: The falling tetromino is shifted towards the center two times to fix the conflict.

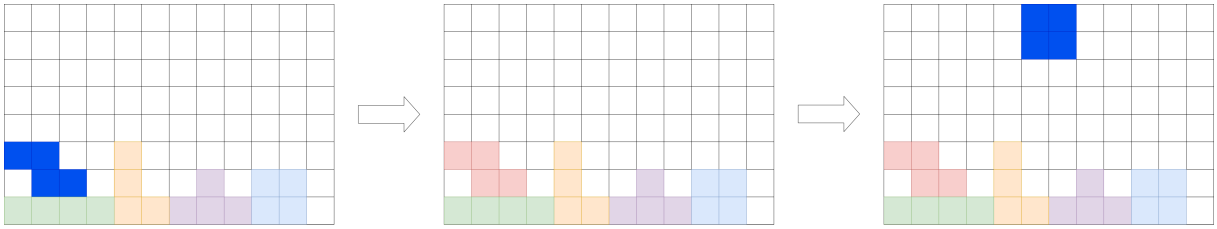


Figure 9: Collision example.
Left: The falling tetromino has a So_COL collision. **Middle**: A collision is detected and the tetromino stops falling. **Right**: A new falling tetromino is generated.

line, removes it, and moves down all pixels on the top. If more than one full line is formed, they all need to be removed, **one by one**. Additionally, we add a blinking effect when removing the lines, so that the game is more attractive to play. The exact blinking behavior will be defined in section 11.2.

2.9 Falling tetromino policy

A falling tetromino moves one pixel downwards approximately every second; in other words, the falling speed of a teromino is 1 pixel/s. Besides falling, the tetrominos should be able to rotate and to move horizontally while falling:

- if a move/rotate button is pressed before the tetromino is moved downwards, the falling tetromino should be moved/rotated accordingly before being moving downwards;
- if the `reset` button was pressed before the tetromino is moved downwards, only the reset operation should be performed;
- if more than one button is pressed, the one that occupies the LSB in the button register determines what action to make (more details in Section 10).

Per game rules, these transformations should happen faster than the falling rate of the tetromino. In our case, there can be at most 5 transformations between two downwards movements. To achieve that, the state of buttons should be checked approximately every 0.2 seconds. The exact details on how time will be handled in assembly will be provided in Section 3.3.

No new tetromino should be generated before the currently falling tetromino has touched something below it and, as a consequence, his location has become fixed. Once this has occurred, on the next down movement of the now fixed tetromino, a new falling tetromino should be generated and the current fixed tetromino should be placed. An example of this can be seen in Figure 9. Note that the full horizontal line removal should be completed **before** the new tetromino is generated.

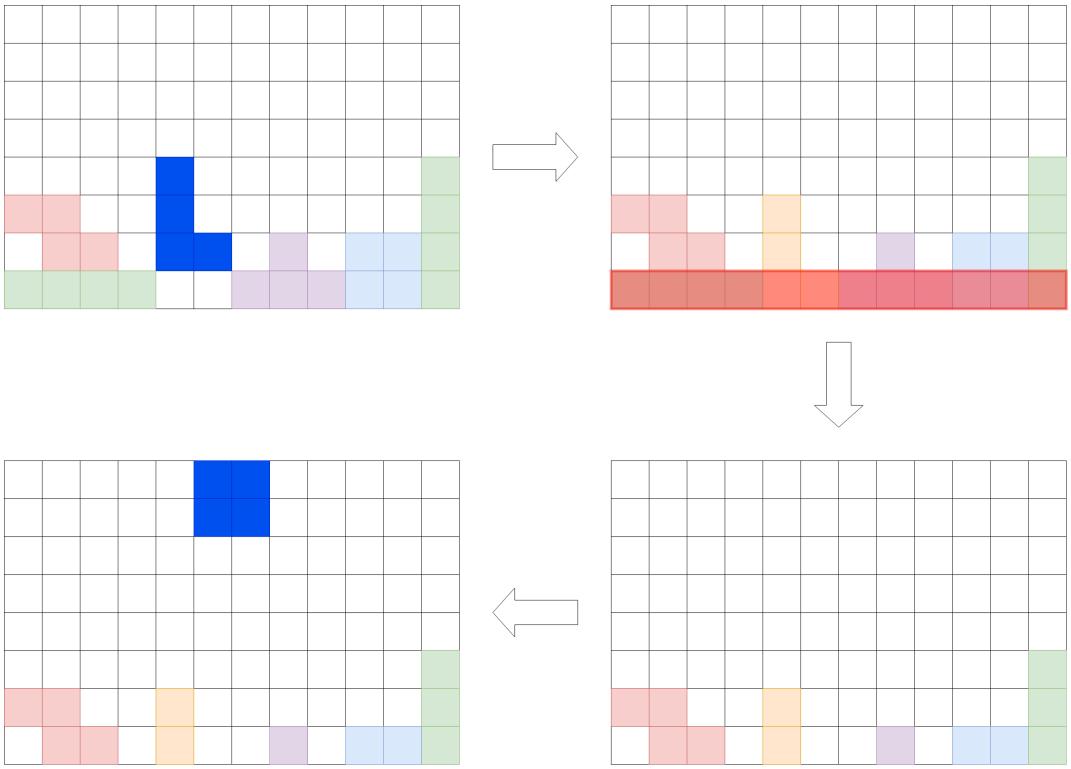


Figure 10: Horizontal line removal
Top Left: Falling tetromino not yet down **Top Right**: Collision So_COL is detected, when moving the falling tetromino down **Bottom Right**: Full line is removed **Bottom Left**: New falling tetromino is generated

2.10 Memory layout

Table 5 shows the precise RAM memory layout for the game. Please keep **exactly** this memory layout as otherwise you will lose points from the automated grading. You are free to use any of the nondesignated locations for your own purpose.

0x1000	**T_X**: Falling tetromino anchor point x-coordinate
0x1004	**T_Y**: Falling tetromino anchor point y-coordinate
0x1008	**T_type**: Falling tetromino shape (type)
0x100C	**T_orientation**: Falling tetromino orientation
0x1010	**Game score**
0x1014	Game State Array (GSA)
0x1018	containing 96 words (32b each)
...	...
0x1198	SEVEN_SEGS[0]
	SEVEN_SEGS[1]
	SEVEN_SEGS[2]
	SEVEN_SEGS[3]
...	...
0x2000	LEDS[0]
	LEDS[1]
	LEDS[2]
...	...
0x2010	RANDOM_NUM
...	...
0x2030	BUTTONS
0x2034	
...	...

Table 5: RAM memory organization for keeping the current state of the game.

We will now present the functions required to run the game, which you will have to implement.

3 Drawing Using the LEDs

Your first exercise is to implement the following two procedures for controlling the LEDs, and a procedure to add execution delay:

1. `clear_leds`, which initializes the display by switching off all the LEDs,
2. `set_pixel`, which turns on a specific LED, and
3. `wait`, which creates an execution delay.

The LED array has 96 pixels (LEDs). Figure 11 translates the pixel **x-** and **y-** coordinate into a 32-bit word and a position of the bit inside that word (0 – 31). The words LEDS[0], LEDS[1], and LEDS[2] are stored consecutively in memory as illustrated in Table 5. As you are accustomed to, the most significant bit of a byte bears the highest index, e.g. 0-th bit is the rightmost and 7-th is the leftmost bit. Bytes are stored in memory in little endian fashion.
Next two sections describe these three procedures. Section 3.4 describes the steps to follow in this exercise.

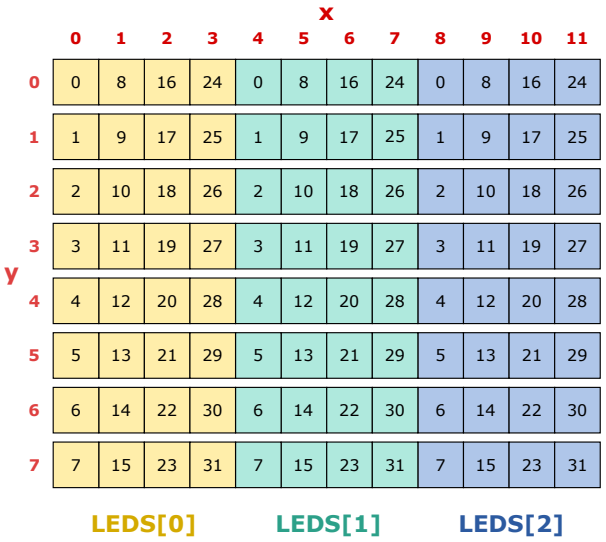


Figure 11: Translating the LED **x** and **y** coordinates into the corresponding bit in the LED array. For example, **x** = 5 and **y** = 3 correspond to the bit 11 in the word LEDS[1].

3.1 Procedure clear_leds

The `clear_leds` procedure initializes all LEDs to 0 (zero). You should call `clear_leds` before drawing new tetromino position.

3.1.1 Arguments

- None

3.1.2 Return Values

- None.

3.2 Procedure set_pixel

The `set_pixel` procedure takes two coordinates as arguments and turns on the corresponding pixel on the LED display. When this procedure turns on a pixel, it must keep the state of all the other pixels **unmodified**.

3.2.1 Arguments

- register `a0`: the pixel's **x**-coordinate.
- register `a1`: the pixel's **y**-coordinate.

3.2.2 Return Values

- None.

3.3 Procedure wait

The `wait` procedure serves to add a delay to the execution of the program. This delay can be created by initializing a very large counter value in a register and decrementing it in a loop. This way, the

execution time needed to decrement the counter to zero will create a time delay. A good value for the delay is approximately 0.2s, and for the Gecko4EPFL board this can be done using an initial counter value of 2^{20}.

Beware that when simulating the assembly program in **nios2sim**, the `wait` procedure can cause the simulation to run too slow. In this case, it is best to either test the `wait` procedure directly on Gecko4EPFL, or to significantely reduce the initial value of the counter for simulation.

3.3.1 Arguments

- None.

3.3.2 Return Values

- None.

3.4 Exercise

- Create a new `tetris.asm` file.
- Implement the `clear_leds`, `set_pixel` and `wait` procedures.
- Implement a `main` procedure that first calls the `clear_leds` to initialize the display, and then calls `set_pixel` to turn on some pixels.
- Simulate your program in the program using the **nios2sim**.
- Implement a new `main` procedure that will make a LED blink on and off approximately every 0.2 s. This can be done using the three functions implemented in this Section.
- To run the program on the Gecko4EPFL, follow the instructions in Section 15.

4 GSA-Related Procedures

Before going further, let us define here some functions that could be of use for the rest of the game:

- `in_gsa`: checks if the given location is a valid GSA location
- `get_gsa`: gets an element from the GSA
- `set_gsa`: sets an element of the GSA

4.1 Procedure in_gsa

This procedure gets as argument a location (x, y) and reports whether this location is inside the GSA or not. In other words, it returns 1 when $x < 0$, $x > 11$, $y < 0$, $y > 7$, and 0 otherwise.

4.1.1 Arguments

- register `a0`: pixel's x-coordinate
- register `a1`: pixel's y-coordinate

4.1.2 Return Value

- register `v0`: 1 if out of GSA, 0 if in GSA

4.2 Procedure get_gsa

This procedure gets as argument a location (x, y) and returns the value p of the element at location (x, y) in the GSA. The function assumes that x and y are in the valid range and does not check them. The value p is in the set: $p \in \{$**NOTHING, PLACED, FALLING**$\}$ (see Section 2.4 and Table 3).

4.2.1 Arguments

- register `a0`: pixel's x-coordinate
- register `a1`: pixel's y-coordinate

4.2.2 Return Value

- register `v0`: Element at location (x, y) in the GSA

4.3 Procedure set_gsa

This procedure gets as argument a location (x, y), a value p, and sets the location (x, y) in the GSA to p. The function assumes that x and y are in he valid and does not check them. The value p is in the set: $p \in \{$**NOTHING, PLACED, FALLING**$\}$ (see Section 2.4 and Table 3).

4.3.1 Arguments

- register `a0`: pixel's x-coordinate
- register `a1`: pixel's y-coordinate
- register `a2`: pixel's value p

4.3.2 Return Values

- None.

5 From the GSA to the LEDs

We now have some ways of setting GSA elements, but we want something to be on the LEDs so that we are able to actually see something. The `draw_gsa` procedure should take what is in the GSA and make the LED display consistent with it. In other words, if GSA element is

- **NOTHING**, the corresponding LED should be off
- otherwise, the corresponding LED should be lit (set to 1)

5.1 Procedure draw_gsa

5.1.1 Arguments

- None

5.1.2 Return Value

- None

5.2 Exercise

- Implement the `draw_gsa` procedure.
- Set some locations in the GSA with values in {**NOTHING, PLACED, FALLING**}, and then call `draw_gsa` for visualisation.
- Simulate the program using the **nios2sim** simulator.

6 Drawing Tetrominoes

Now that something can be displayed on the LEDs, let's draw something meaningful. This is the role of the `draw_tetromino` procedure, which is in charge of drawing the tetrominoes.

A simple implementation would hardcode every possible orientation of each tetromino; however, this is error-prone and the function will be very long. We will show you how you can encode the same information in a suitable data structure instead, which will allow you to write a much simpler and cleaner code for `draw_tetromino`.

6.1 Procedure draw_tetromino

`draw_tetromino` should read **T_X, T_Y, T_orientation, T_type** from the RAM and should set the corresponding GSA elements to the correct value. We will pass the new value for the tetromino GSA elements as a function argument: this will allow us to use `draw_tetromino` not only to draw the falling tetromino (by writing **FALLING**) but also to erase it just before moving it (**NOTHING**) or to set it as **PLACED** once it reaches the bottom of the screen. In the following, we will refer to all of these operations as *drawing* the tetromino, even when we actually removing it or placing it.

6.1.1 Arguments

- register `a0`: GSA value p, such that $p \in$ {**NOTHING, PLACED, FALLING**}

6.1.2 Return Value

- None

In Section 2.1 we defined the anchor point as "the reference point for a tetromino", such that "we draw tetrominoes with respect to the location of their anchor points". More specifically, we can observe that the shape of every tetromino orientation is fully defined by an *array of three (x, y) pairs*, which contain the offsets of the three non-anchor points with respect to the anchor point. For example, the L tetromino facing east (Figure 18(b)) is fully defined by the following array: {(0, -1), (0, 1), (1, 1)} and the B tetromino facing north (Figure 15(a)) by {(-1, 0), (1, 0), (2, 0)}. The pixels occupied by a tetromino can then be reconstructed by adding these offsets to the coordinates of the anchor point, stored in **T_X, T_Y**; finally, drawing a tetromino means updating those pixels together with the anchor point. Going back to the previous example of the L tetromino facing east, if (**T_X, T_Y**) = (4, 2), then the pixels it occupies are {(4, 1), (4, 3), (5, 3), (4, 2)}.

The idea is therefore to store all tetromino orientations as arrays of six offsets (three for x and three for y) so that `draw_tetromino` will only have to go through these arrays, compute the occupied pixels as described above and modify them in the GSA according to the `a0` argument. To initialize the arrays in ROM, we will use the `.word` directive, which allocates a 32-bit word in memory, followed by the value that we want to store in that word. To locate that word in memory, we prepend a label which we can use in the program wherever an immediate is expected: the assembler will eventually replace all labels with the address of the word in memory. Multiple consecutive `.word` statements will be stored consecutively in memory, which means that the respective value can be retrieved by summing an offset to the label. This allows us to define arrays of multiple 32-bit words. For example, the following code snippet will write 42 to LEDS and 167 to LEDS+4:

```
1 ldw t0, my_const(zero)
2 stw t0, LEDS(zero)
3 addi t1, zero, 4
4 ldw t2, my_const(t1) ; second value of the my_const array at address my_const+4
5 stw t2, LEDS+4(zero)
6 break
7 ; the array my_const is in the next page
```

```
8  my_const:
9  .word 42
10 .word 167
```

We defined and initialized for you the tetromino arrays in the template (**DRAW_Ax** and **DRAW_Ay** for x and y offsets respectively). Check carefully how they have been organized in memory as this will define how you can use **T_type** and **T_orientation** to retrieve the respective x and y offset arrays. Note that **DRAW_Ax** and **DRAW_Ay**, as well as the way they are used in `draw_tetromino`, may also be useful in some of the following functions as well.

7 Tetromino Generation

Now that we have some way of drawing the tetrominoes, we can implement the procedure that generates them: `generate_tetromino`.

7.1 Procedure generate_tetromino

This procedure takes no argument. It firsts reads a random number from the random number generator in order to pick new tetromino randomly. The random number generator produces new random 32b value every time one reads from it. Note that the random value is a 32b number and here we only need to have randomness over 5 distinct tetrominoes. You should take the last 3 bits and, if the corresponding number is not between 0 and 4, redraw a new number from the random number generator. This should only take a few tries, which is not a problem as a new random number is generated every cycle. The mapping has to be the one suggested by the tetromino enumeration, previously defined in Table 1: if the random generator gives 0, a C should be generated, if it gives 1, a B should be generated, and so on.

Then, once the shape of new tetromino is determined, the tetromino should be placed at the default anchor-point location: **(6,1)**oriented northwards. In order to do this, the tetromino description in both the GSA and the tetromino structure (**T_X, T_Y, T_orientation, T_type**) should be initialized accordingly.

To simulate random number generation in **nios2sim**, you can write a value of your choice at the location RANDOM_NUM. Inside the **nios2sim** this memory location overlaps with the address space reserved for UART (Figure 12); this poses no issues as your processor does not use UART. Therefore, to access the location RANDOM_NUM in nios2sim, open the UART tab and look for the field `receive`.

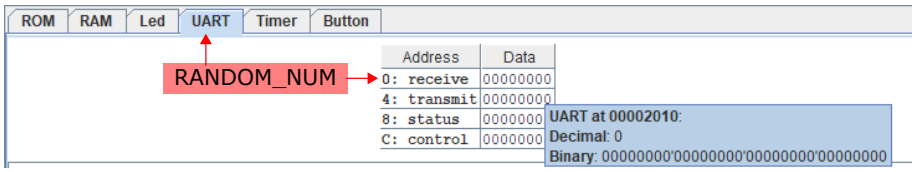


Figure 12: The location where you should write a random number for simulating the game in nios2sim.

7.1.1 Arguments

- None

7.1.2 Return Value

- None

8 Collision Mechanisms

One of the yet unknown aspects of the game is the collision detection, which is taken care of by the `detect_collision` procedure.

8.1 Procedure detect_collision

As the direction of the movement determines the collision type that can be signaled and vice versa, this procedure takes as argument only the collision type we want to check for (see Table 4). It then checks if the specified collision should be signled. If yes, the procedure returns the same value it received at the input. If no, it returns NONE (see Table 4). When completed, the procedure should leave tetromino structure unchanged. Based on the input value, this procedure separates two important cases:

- In the case of E_COL, W_COL and S_COL, the procedure checks for the collision by checking if a tetromino will overlap with an obstacle if moved in an intended direction. The easiest way of detecting if a collision should be signaled is to try to read from the game state array exactly at the location the tetromino intends to occupy after the corresponding movement. If the GSA is not empty this procedure signals the collision by returning the value that was passed in the argument.
- In case of an OVERLAP collision, the procedure should act a bit differently. Since the OVERLAP collision can be a result of a rotation of the tetromino (left or right) or a generation of a new tetromino, it is not possible to determine the exact cause of the collision just by knowing that the input argument is OVERLAP. In that case, the procedure assumes that the tetromino has already been rotated or generated (i.e. the tetromino structure has been changed), and it only checks if an overlap currently exists.

8.1.1 Arguments

- register `a0`: value of collision we are inquiring about, from Table 4.

8.1.2 Return Value

- register `v0`: same as `a0` if the collision exists indeed, NONE otherwise

9 Moving Tetrominoes

Now that we know how to generate and draw a falling tetromino, we want to be able to move it. Here, we describe the functions that will enable us to do this:

- `rotate_tetromino`: rotate the falling tetromino according to the given orientation. This procedure is called internally by `act`. Please include each of them inside their own BEGIN ... END construct to ensure correct grading.
- `act`: move the falling tetromino according to the argument value:
  - if moveD, moveL, or moveR, move the tetromino down, left, right, respectively;
  - if rotR or rotL, rotate the tetromino clockwise or counterclockwise, respectively.

  This procedure handles the reset case as well.

9.1 Procedure rotate_tetromino

This procedure should change the tetromino orientation according to the given direction d, where $d \in rotR, rotL$ from Table 6, by changing the **T_orientation** attribute in RAM memory. The orientation can only be changed in steps of 90°.

9.1.1 Arguments

- register `a0`: Rotation direction

9.1.2 Return Value

- None

9.2 Procedure act

This functions gets as argument one action, where the actions are defined as symbol/value pairs in Table 6. It then tries, depending on the value received, to do the requested action with the tetromino, after having performed some collision detection. If it fails, it leaves GSA, **T_X**, **T_Y**, and **T_orientation** unchanged.

value	action
0x01	moveL
0x02	rotL
0x04	reset
0x08	moveR
0x10	rotR
0x20	moveD

Table 6: List of symbol/value pairs representing actions to apply over a tetromino.

Depending on the argument value, the procedure should do as follows:

- moveD: If there is no collision at the bottom side of the falling tetromino, it should **add 1** to **T_Y** (Table 5).

- moveL: If there is no collision on the left of the falling tetromino, it should **remove 1** from **T_X** (Table 5.
- moveR: If there is no collision the right of the falling tetromino, it should **add 1** to **T_X** (Table 5.
- rotR (rotL): it should **add 90°(-90°)** to **T_orientation** (Table 5). Then, it should check for a potential OVERLAP collision. If there is one, it should try to move the tetromino towards the center, for at most two pixels. If even after that there exists an OVERLAP collision, it should leave the GSA, **T_X**, **T_Y**, and **T_orientation** unchanged.
- reset: it should call the `reset_game` procedure.

9.2.1 Arguments

- register `a0`: value of the action required (Table 6).

9.2.2 Return Value

- register `v0`: 0 if succeeded, 1 if failed

10 Inputs to the Game

Interacting with the game is the responsibility of `get_input` procedure, which reads the state of the push buttons and returns it, so that the rest of the game, in particular `act` procedure can apply the actions required.

The five push buttons of the Gecko4EPFL are read through the **Buttons** module, which are memory mapped (see Table 5). This module has two 32-bit words, called `status` and `edgecapture`, described in Table 7. To implement `get_input`, you will need to use `edgecapture` (you can ignore `status`).

Address	Name	31 . . . 5	4 . . . 0
`BUTTONS`	`status`	*Reserved*	State of the Buttons
`BUTTONS+4`	`edgecapture`	*Reserved*	Falling edge detection

Table 7: The two words of the **Buttons** module.

The `status` contains the current state of the push buttons: if the bit at the position i is 1, the button i is *currently* released, otherwise (when $i = 0$) the button i is *currently* pressed.

The `edgecapture` contains the information whether the button i ($i = 0, 1, 2, 3, 4$) was pressed. If the button i changed its state from released to pressed, i.e. a falling edge was detected, `edgecapture` will have the bit i set. The bit i stays at 1 until it is explicitly cleared by your program. Mind that when you attempt to write something in `edgecapture`, regardless of the value **the entire** `edgecapture` **will be cleared**; there is no possibility to clear its individual bits.

In the **nios2sim** simulator, you can observe the behavior of buttons module by opening the **Button** window and clicking on the buttons. In the simulator, the buttons are numbered from 0 to 4.

10.1 Procedure get_input

`get_input` should check which button was pressed and, based on that, return the corresponding action value (see Table 6). Do not forget to clear edgecapture once done.

10.1.1 Arguments

- None

10.1.2 Return Value

- register `v0`: action corresponding to the pressed button or 0 if no button is pressed.

In case multiple buttons are pressed simultaneously, the least significant bit of `status` which is set needs to be considered as the input.

11 Handling Full Lines

To handle full lines, let us implement two functions: one that detects if a full line exists and one that removes it from the game screen.

- `detect_full_line`: detect if one line is full in the game state array;
- `remove_full_line`: remove one full line from the game state array, and move the tetrominoes stacked above this line one pixel down.

11.1 Procedure detect_full_line

This procedure returns the smallest y such that GSA$(x, y) = 1\ \forall x \in \{0, 1, ..., 11\}$. If no such y exists, i.e., if $\forall y \in \{0, 1, ..., 7\}, \exists x \in \{0, 1, ..., 11\}$ such that GSA$(x, y) = 0$, the procedure should return $y = 8$, the smallest y-coordinate that is too high to fit the game screen.

11.1.1 Arguments

- None

11.1.2 Return Value

- register `v0`: y-coordinate of the full line closest to the top of the game screen or 8, if there are no full lines.

11.2 Procedure remove_full_line

This procedure takes as input the y-coordinate of a full line. It first makes the entire line blink and then removes it. When removing the full line, it makes every pixel that is above the line move for one pixel down. For moving down the line, this procedure does not call `draw_gsa`. It just updates the gsa array. This is not the case for implementing blinking lines because we need to call `draw_gsa` to really turn the leds on and off.

The line blinking should remove and redraw the lines according to the following sequence: turn LEDs off, on, off, on, and finally off. Remember to call the wait function in between so that human eyes can detect the blinking.

11.2.1 Arguments

- register `a0`: y-coordinate of the full line to be removed

11.2.2 Return Value

- None

12 Game Score

When a full line is detected, not only that we need to remove it, we need to increment the score as well. The maximum score for this game is 999. Score management is done using the following two procedures:

- `increment_score`: increasing the score by one;
- `display_score`: show the score on the 7-segment display.

12.1 Procedure increment_score

This procedure increments the score (increases by 1), every time it is called. Game score is kept in RAM memory, see Table 5 and Section 1.2 for details.

12.1.1 Arguments

- None

12.1.2 Return Value

- None

12.2 Procedure display_score

This procedure displays the current score on the 7-segment display. Game score is kept in RAM memory, see Table 5 and Section 1.2 for details. Similarly, 7-segment displays are also mapped to RAM memory, as described in Table 5 and Section 1.2. You can find which memory location maps to which 7segment digit in figure 13.

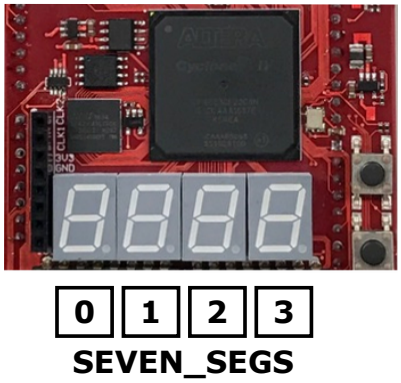


Figure 13: Seven segment displays.

In order to display one decimal figure on one 7-segment display, you can use the following array, which is already specified for you in the template:

```
font_data:
        .word 0xFC  ; 0
        .word 0x60  ; 1
        .word 0xDA  ; 2
        .word 0xF2  ; 3
        .word 0x66  ; 4
```

```
7           .word 0xB6  ; 5
8           .word 0xBE  ; 6
9           .word 0xE0  ; 7
10          .word 0xFE  ; 8
11          .word 0xF6  ; 9
```

In this array, a value at index i corresponds to the code which when displayed on 7-segment display results in decimal digit i.

12.2.1 Arguments

- None

12.2.2 Return Value

- None

13 Reset

After all, we need a function that resets the game to its default state: `reset_game`.

13.1 Procedure reset_game

This procedure, if called, puts the game in its default state. The default state is defined as follows:

1. The game score is zero. All four 7-segment displays show zero.
2. A new tetromino is generated and placed with its anchor point at the initial location **(6,1)**.
3. All the remaining game state array locations are initialized zero.
4. The LEDs are lit according to the game state array.

13.1.1 Arguments

- None

13.1.2 Return Value

- None

14 Putting Everything Together

Now that we have all the functions required to run the game, we should implement the main game loop that calls the functions in the good order. This is actually implementing the `main` function. The `main` function will not be tested by the automated grader but you need to implement it in order to do the demonstration of the game.

To control how many game loop iterations we allow for moving left/right or rotating the tetromino, without moving it downwards, we define a symbol called RATE (Section 1.2). Your game should move the falling tetromino down only after every RATE number of iterations.

Here is a detailed explaination for the game:

Algorithm 1: Tetris game

```
begin
    RATE ← 5
    reset the game
    repeat
        repeat
            i ← 0
            while i < RATE do
                draw the GSA on the leds and display the score
                remove the falling tetromino from the GSA, not from the tetromino structure
                wait approximately 0.2 s
                get the button input
                if a button has been pressed, try to do the required action, if possible (act, etc...)
                redraw the falling tetromino on the GSA
            end
            remove the falling tetromino from the GSA, not from the tetromino structure
            try to move the falling tetromino down
            redraw the falling tetromino on the GSA
        until falling tetromino can't be drawn when moving down
        replace the falling tetromino by a placed tetromino
        while ∃ a full line do
            remove the bottommost full line
        end
        generate a new tetrominoe
        detect for overlaping collisions
        if no collisions then
            draw the falling tetromino on the GSA
        end
    until newly generated falling tetromino overlaps with something
end
```

15 Playing the Game

Now that you have implemented all the required core functionality, it is time to test if the game runs smoothly end to end. You can do this by simulating your program in the **nios2sim** simulator. A better way to do this is to run the program on the Gecko4EPFLboard.

The steps necessary to run the program on the **Gecko4EPFL** board are the following:

- Please use the **Nios II** CPU provided in the Quartus project `quartus/GECKO.qpf` in the template.
- When playing the game on Gecko4EPFL, make sure that you add a call to `wait` after drawing in

the main, and that `restart_game` is called whenever the game is lost. Otherwise it may happen that a press on the reset button is ignored by your program.

- In the **nios2sim** simulator, assemble your program (Nios II → Assemble) and export the ROM content (File → Export to Hex File → Choose ROM as the memory module) as [template folder]/quartus/ROM.hex. **Do not modify anything else in the Quartus project folder.**
- Compile the Quartus project.
- Program the **FPGA**.

Every time you modify your program, remember to regenerate the Hex file and to compile the Quartus project again before programming the FPGA.

While implementing the **TETRIS** game, you might have come across design choices that are not addressed specifically or left unclear in this document. For those cases, you can safely assume that whichever choice you deem fit will be considered as valid and will not result in loss of points in the final grading.

16 Submission

You are expected to submit your complete code as a single .asm file. The automatic grader will look for and test the following procedures: `clear_leds`, `set_pixel`, `get_input`, `set_gsa`, `move_gsa`, `in_gsa`, `display_score`, `reset_game`, `detect_full_line`, `remove_full_line`, `increment_score`, `act`, `rotate_tetrominoe`, `detect_collision`, `draw_gsa`, `draw_tetromino`, and `generate_tetrominoe`.Make sure that you follow the formatting instructions detailed in Section 1.3.

Each of the above listed procedures is tested independently of the rest of your code; everything **around** the tested procedure the grader will replace with the default code. Therefore, you must enclose between appropriate comments all the auxiliary undeclared procedures your code calls (see Section 1.3).

There are two submission links: **tetris-preliminary** and **tetris-final**:

- You can use the preliminary test as many times as you wish until the deadline. The preliminary tests only checks if the grader found and parsed correctly all the procedures and if your assembly code compiles without errors. The preliminary feedback will refer to these checks only.
- The final test will assess the correctness of the procedures enlisted above by analyzing their effect on memory contents and registers. The final feedback will resemble the following: Procedure procedure_name passed/failed the test.

If your code passes all the tests in **tetris-final**, you will obtain the maximum score of 80%. For the remaining 20%, you will need to make a successful live demonstration of the game on Gecko4EPFL to the teaching assistants.

You are allowed (and encouraged) to add other features to the game. However, you must not submit an enhanced game to the automated grader, as it expects a basic game that conforms to the instructions detailed in this document. Teams that demonstrate the most interesting and complete game will have a chance to win **the ArchOrd Christmas gift**! So, be creative!

A Tetrominoes Orientation

To illustrate tetromino rotations on an example, let us assume the anchor point at the location (4,4) and draw all orientations for all tetrominoes below.

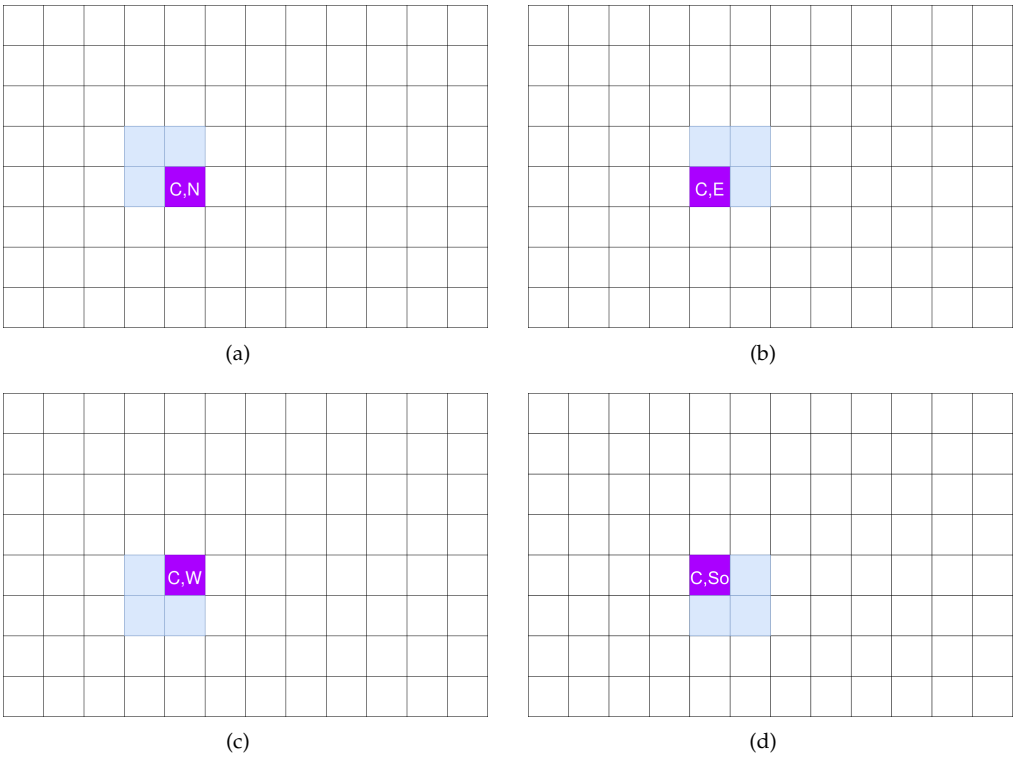
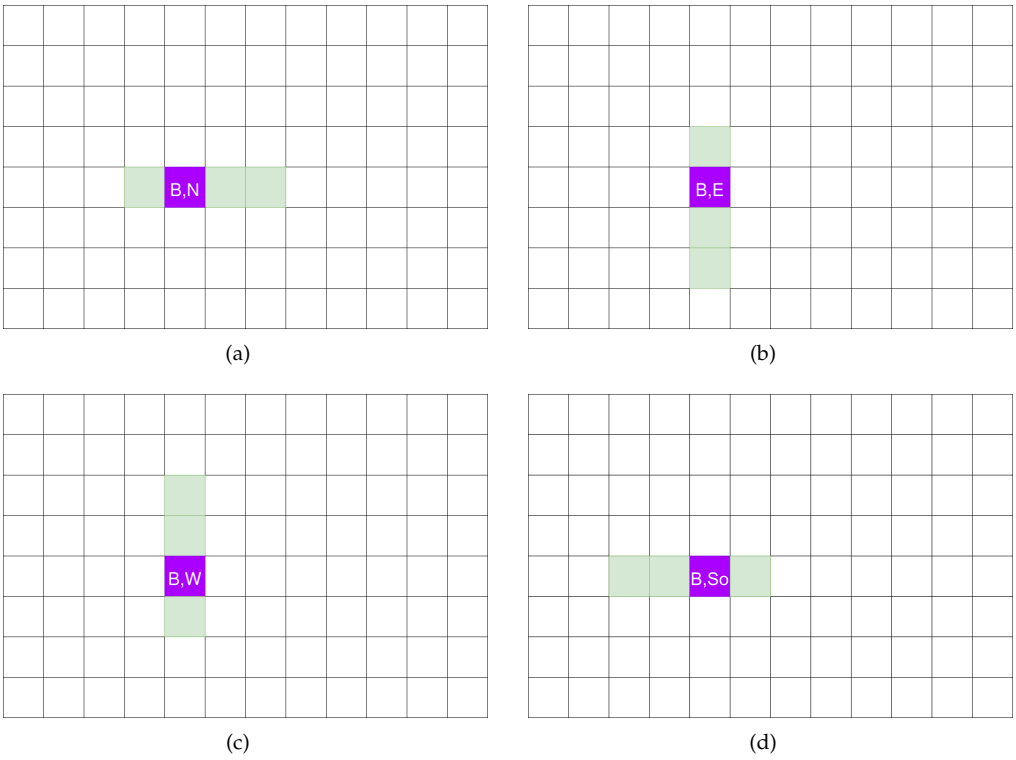
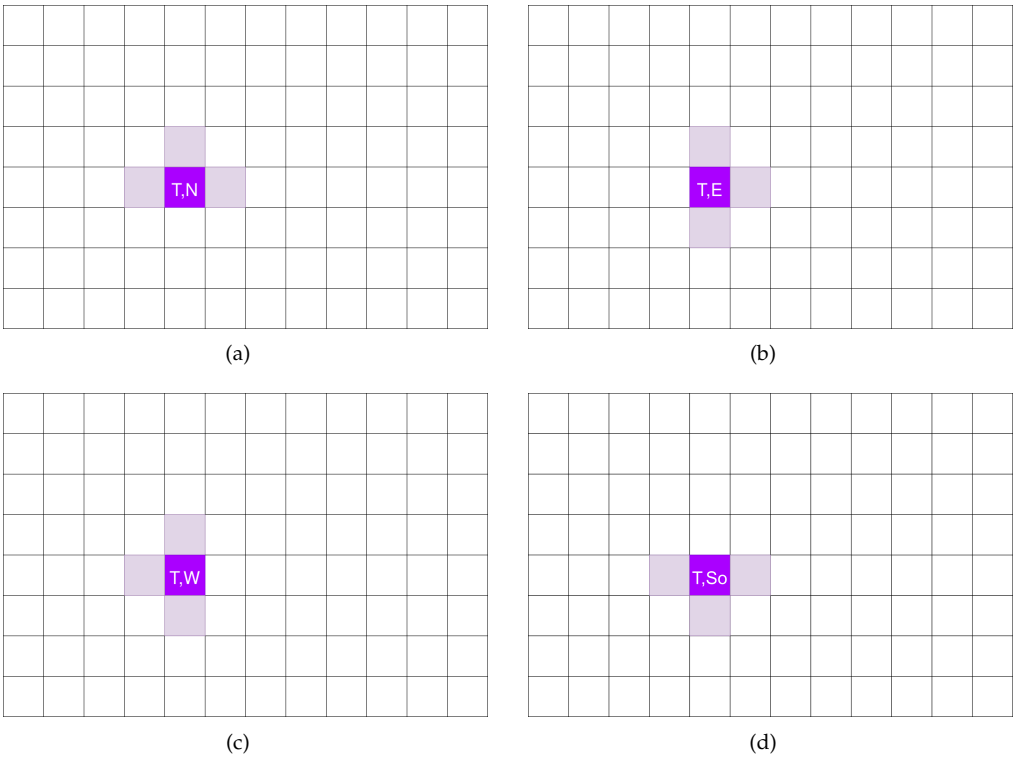


Figure 14: Square (C): (a) facing north (N); (b) facing east (E); (c) facing west (W); (d) facing south (So).



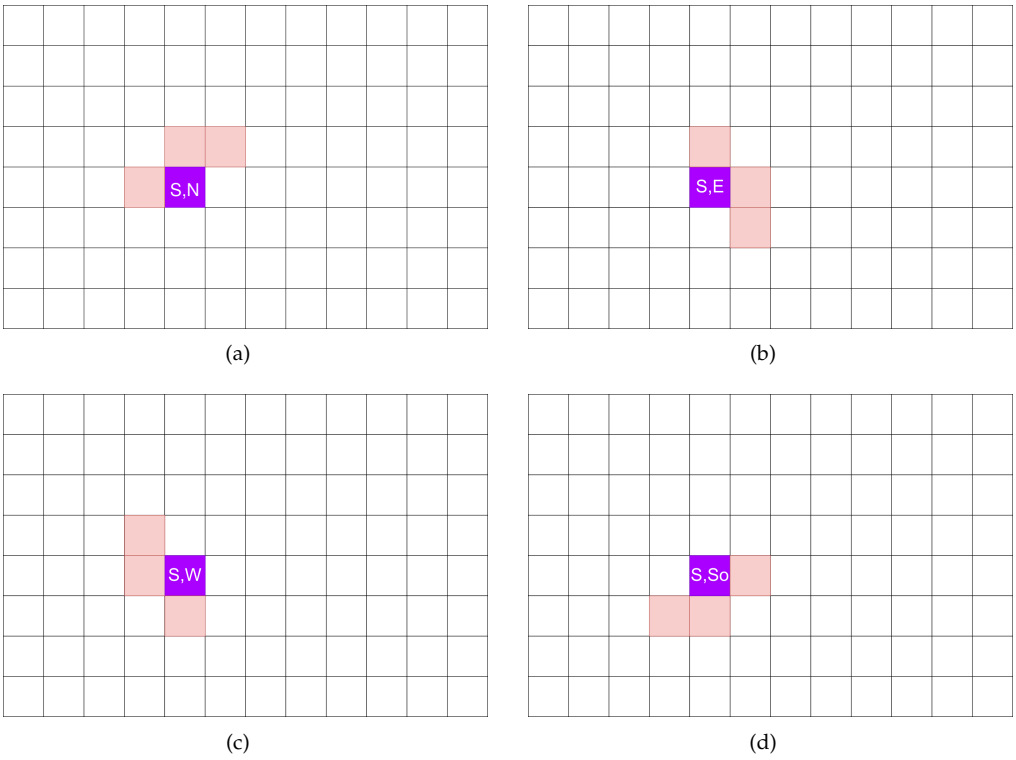
(a) (b) (c) (d)

Figure 15: Bar (B): (a) facing north (N); (b) facing east (E); (c) facing west (W); (d) facing south (So).



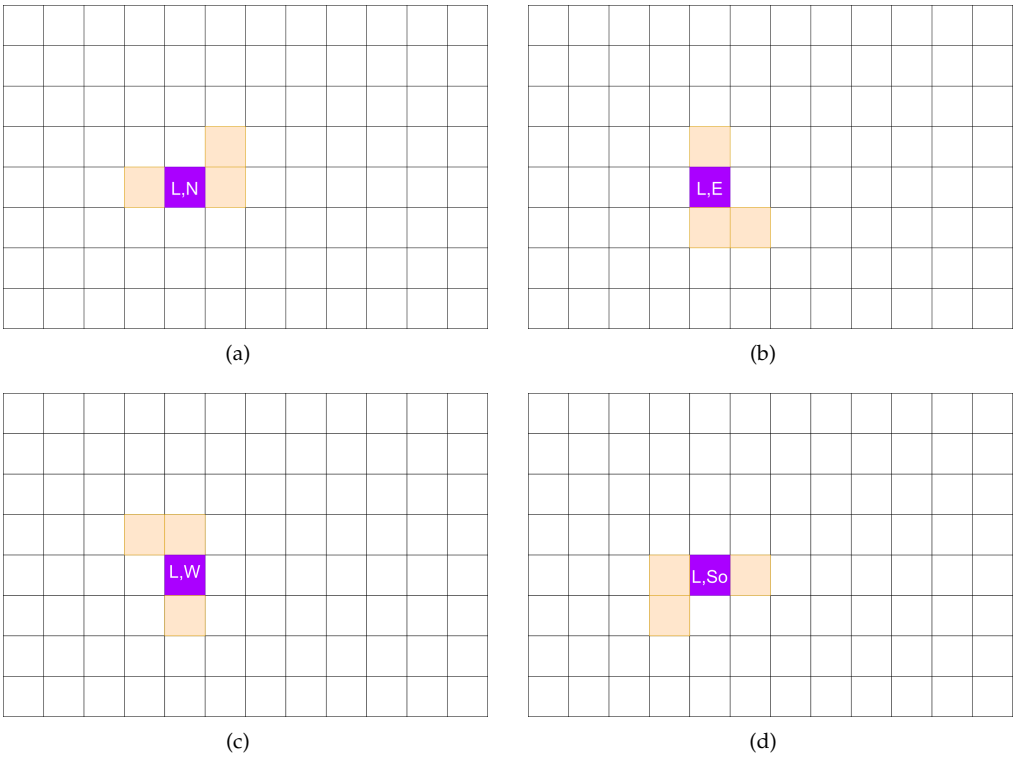
(a) (b) (c) (d)

Figure 16: T-shape: (a) facing north (N); (b) facing east (E); (c) facing west (W); (d) facing south (So).



(a) (b) (c) (d)

Figure 17: S-shape: (a) facing north (N); (b) facing east (E); (c) facing west (W); (d) facing south (So).



(a) (b) (c) (d)

Figure 18: L-shape: (a) facing north (N); (b) facing east (E); (c) facing west (W); (d) facing south (So).