CSE379 - Introduction to Microprocessors - Lab 7

Liam Mullen, Marcos De La Osa Cruz
liammull, marcosde
Section R1
4/25/2022

Table of Contents

Section 1:

Liam M:

- Develop subroutines for:
    - Random Number Generation
    - UART Handler
    - Switch Handler
    - Timer Handler
    - Menu Screens
    - ANSI escape sequences
- Created github repo, and defined constants for data and text sections

Marcos D:

- Develop subroutines for:
    - Block movement
    - Block Merging
    - Rendering & Animation
    - Pointer Management & Game Logic
    - Develop Test Cases/Procedures

Focused on bottom up development, started by developing the pointer abstraction and flowcharts for game logic. Then moved onto rendering the game board as it existed as strings in memory. Finished with developing movement logic and dealing with UART interrupt based movement system.

Collaboratively, we developed the game logic, library file, general program structure, and debugged the program.

Section 2:

Game Board Abstraction

SQ0 2	SQ1 0	SQ2 4	SQ3 8
SQ4 4	SQ5 0	SQ6 8	SQ7 16
SQ8 8	SQ9 0	SQ10 16	SQ11 32
SQ12 16	SQ13 0	SQ14 32	SQ15 64

Understanding the pointer abstraction layers

Value0	Value1	Value2	Value3
SQ0	SQ1	SQ2	SQ3

Definition: Value is the integer representing the number in a block all powers of 2 between [2^0 - 2^{10}]

Definition: Square (SQ) is the pointer to the location of an individual game board square, they remain constant pointers to location while the game is played. Numbered SQ0-SQ15

Explanation:

While the grid is rendered to the screen as a singular string the values of the grid are not stored in the same string. Instead our group has created a pointer abstraction layer, which corresponds to the grid. In this implementation the individual squares are represented as pointers, which point to their corresponding value.

This implementation lets us use cursor movements based upon the SQ number to output ESC sequences based upon the value of the SQ. (See more information in render_game_board)

In doing this our implementation never alters the original gameboard, instead it overlays the last ESC blocks with the new ones generated from their memory locations.

We decided upon this implementation due to it's ease of debugging. Since we could tell exactly what was placed in what block using the memory browser in CCS.

Render Game Board

Using the stack to simplify Animation

SQ0 2	SQ1 0	SQ2 4	SQ3 8
SQ4 4	SQ5 0	SQ6 8	SQ7 16
SQ8 8	SQ9 0	SQ10 16	SQ11 32
SQ12 16	SQ13 0	SQ14 32	SQ15 64

SQ0

2

Location: move_to_SQ0

Value: SQ0 -> 2

Rendered: Block2

Concept: We can deduce the location and the value of square on the game board by the order that we pop the SQ's off the stack.

Explanation:

In order to increase code density and minimize our complexity, we used the stack to store the SQ pointers. When we render the game board we start by pushing the SQ0-SQ15 pointers to the stack. We push the SQ's in the opposite order that we want to pop them. So we start by pushing SQ15 then push them all in descending order until we reach SQ0.

The benefit of this implementation is that by keeping track of the amount of SQ's that we popped with an accumulator, we get the location of the SQ on the board. Then we can access the value of that SQ to determine the block number that we have to render. Thus we can render the entire gameboard from the pointer abstraction layer. This means that we never have to touch the gameboard grid, or have to do any cursor math

SQ15	64
SQ14	32
SQ13	0
SQ12	16
SQ11	32
SQ10	16
SQ9	0
SQ8	8
SQ7	16
SQ6	8
SQ5	0
SQ4	4
SQ3	8
SQ2	4
SQ1	0
SQ0	2 ←**Accumulator**

As we POP SQ from the stack the accumulator increases in proportion to the SQ number
(EX SQ3 would have accumulator of 3)

POINTER BASED MOVEMENT

Understanding movement based on the pointer abstraction layer

Merge Pointers → SQ Pointers → Values

Definition: Merge Pointer, a pointer to the SQ pointers. They keep track of blocks that have already been merged on the row

Explanation:

Due to the mechanics of 2048, movements will remain in the same row or column that they began in. Our algorithm leverages this fact by separating movements by row if they are left or right, or by column if they are up or down. This lets the algorithm focus on merging only one set of SQ's at a time.(Diagram 1).

Additionally, we keep track of any capped merges that occur with the merge pointer system. Upon the start of the movement all merge pointers (A-H) are assigned a row (diagram 2) or column (diagram 3). Assigned merge pointers keep track of any SQ that contains an already merged block. In the algorithm we then check the merge pointers for that row or column to determine whether or not blocks in that row or column can merge.

Merge pointers persist throughout the whole movement subroutine. They are only reset when the subroutine closes and the current movement direction has been completed.

LEFT ↔ RIGHT

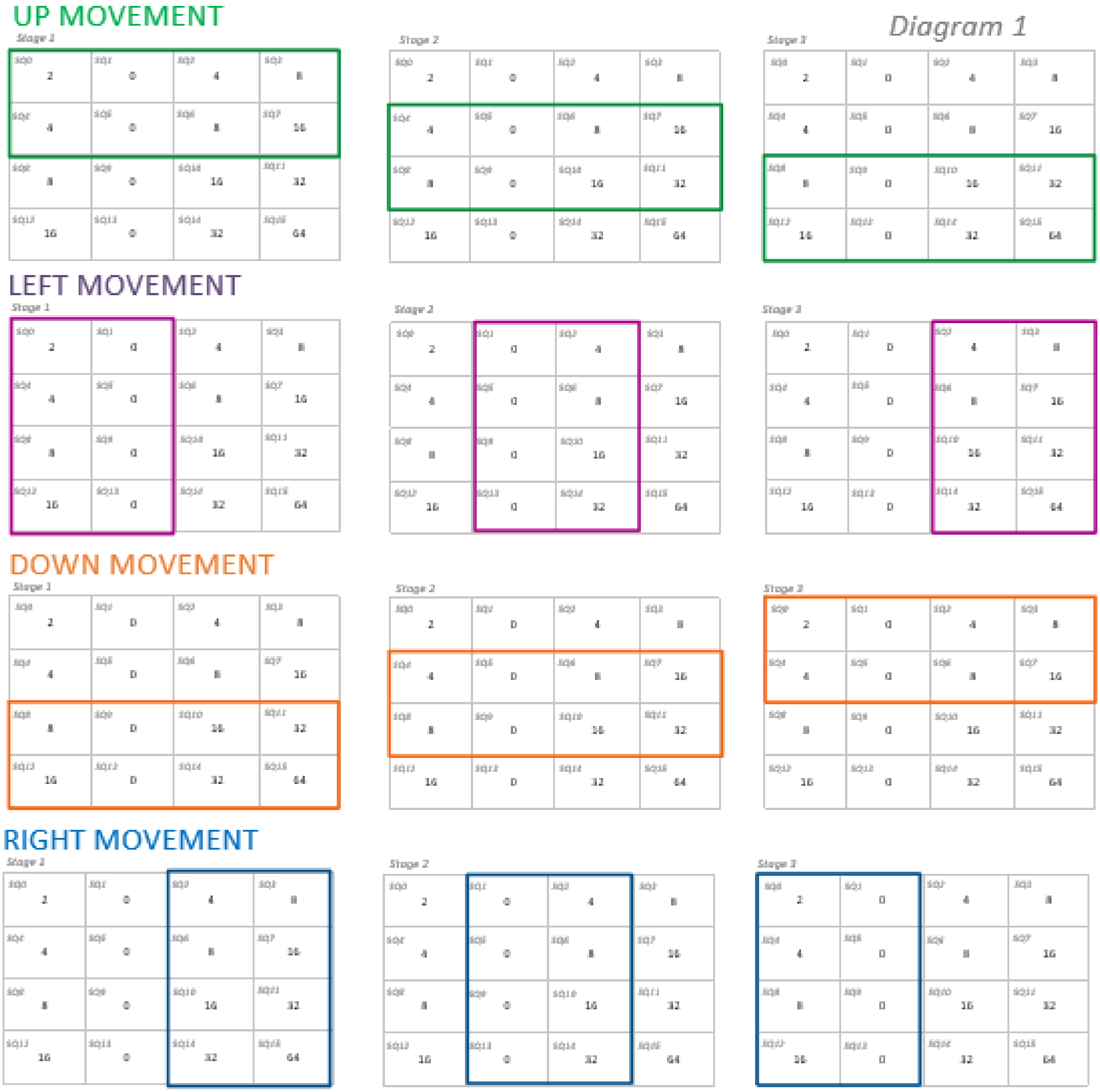
Diagram 2

Merge pointers A & B	*SQ0* 2	*SQ1* 0	*SQ2* 4	*SQ3* 8
Merge pointers C & D	*SQ4* 4	*SQ5* 0	*SQ6* 8	*SQ7* 16
Merge pointers E & F	*SQ8* 8	*SQ9* 0	*SQ10* 16	*SQ11* 32
Merge pointers G & H	*SQ12* 16	*SQ13* 0	*SQ14* 32	*SQ15* 64

UP ↕ DOWN

Diagram 3

SQ0 2	*SQ1* 0	*SQ2* 4	*SQ3* 8
SQ4 4	*SQ5* 0	*SQ6* 8	*SQ7* 16
SQ8 8	*SQ9* 0	*SQ10* 16	*SQ11* 32
SQ12 16	*SQ13* 0	*SQ14* 32	*SQ15* 64
Merge pointers A & B	Merge pointers C & D	Merge pointers E & F	Merge pointers G & H


UP MOVEMENT
Diagram 1
Stage 1
2 0 4 8
4 0 8 16
8 0 16 32
16 0 32 64
Stage 2
2 0 4 8
4 0 8 16
8 0 16 32
16 0 32 64
Stage 3
2 0 4 8
4 0 8 16
8 0 16 32
16 0 32 64
LEFT MOVEMENT
Stage 1
2 0 4 8
4 0 8 16
8 0 16 32
16 0 32 64
Stage 2
2 0 4 8
4 0 8 16
8 0 16 32
16 0 32 64
Stage 3
2 0 4 8
4 0 8 16
8 0 16 32
16 0 32 64
DOWN MOVEMENT
Stage 1
2 0 4 8
4 0 8 16
8 0 16 32
16 0 32 64
Stage 2
2 0 4 8
4 0 8 16
8 0 16 32
16 0 32 64
Stage 3
2 0 4 8
4 0 8 16
8 0 16 32
16 0 32 64
RIGHT MOVEMENT
Stage 1
2 0 4 8
4 0 8 16
8 0 16 32
16 0 32 64
Stage 2
2 0 4 8
4 0 8 16
8 0 16 32
16 0 32 64
Stage 3
2 0 4 8
4 0 8 16
8 0 16 32
16 0 32 64

Example 1

Complete Movement

0	0	2	2
Block 0	Block 1	Block 2	Block 3

Definition: Complete Movement, is referring to the movement of one block from one side of the board to the next

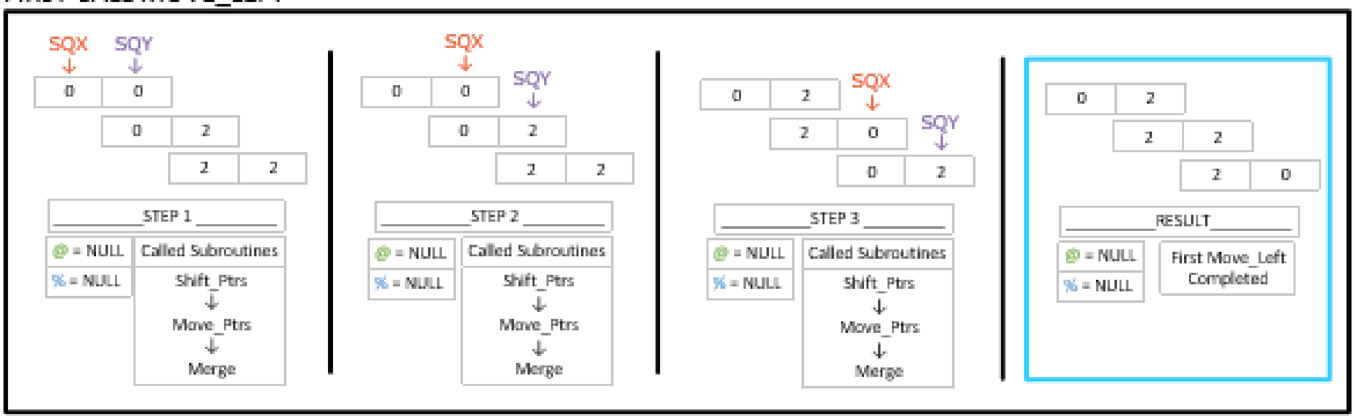
Explanation:

This example demonstrates the basic concept of our move and merge game logic. SQX & SQY are variables that stand for the current two pointers referring to the value of the blocks being moved. While the value of those blocks is stored in the table.

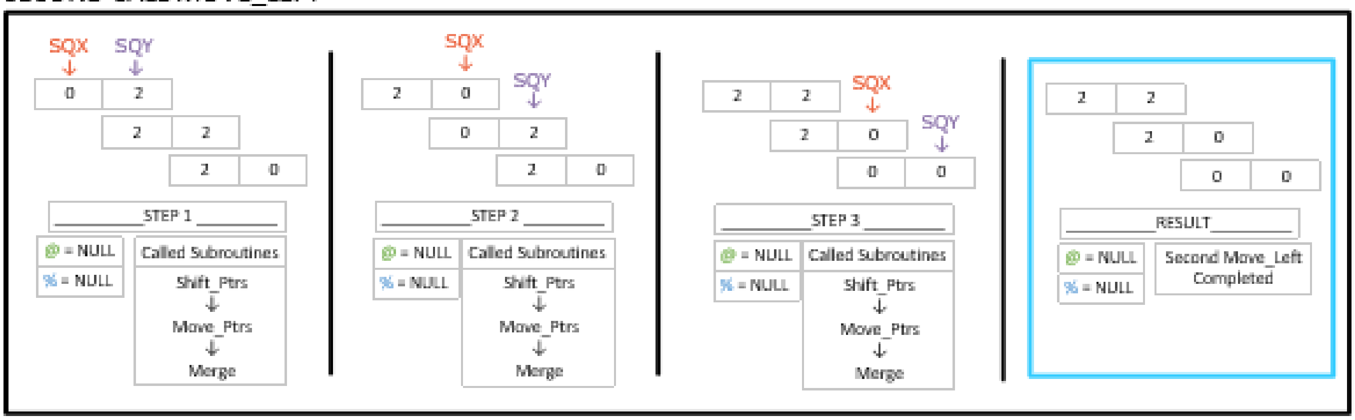
The algorithm we developed merges two blocks at a time starting from the game board direction called (i.e.) Move_Left merges the left most two blocks and continues right. To complete one pass through of the movement logic, the algorithm compares the values at SQX &SQY three times. This is represented by breaking up the 1x4 vector into 3 1x2 vectors in the corresponding diagrams.

The algorithm must be called three times per press of UART keypad, to generate a complete movement. This is demonstrated in the corresponding example.

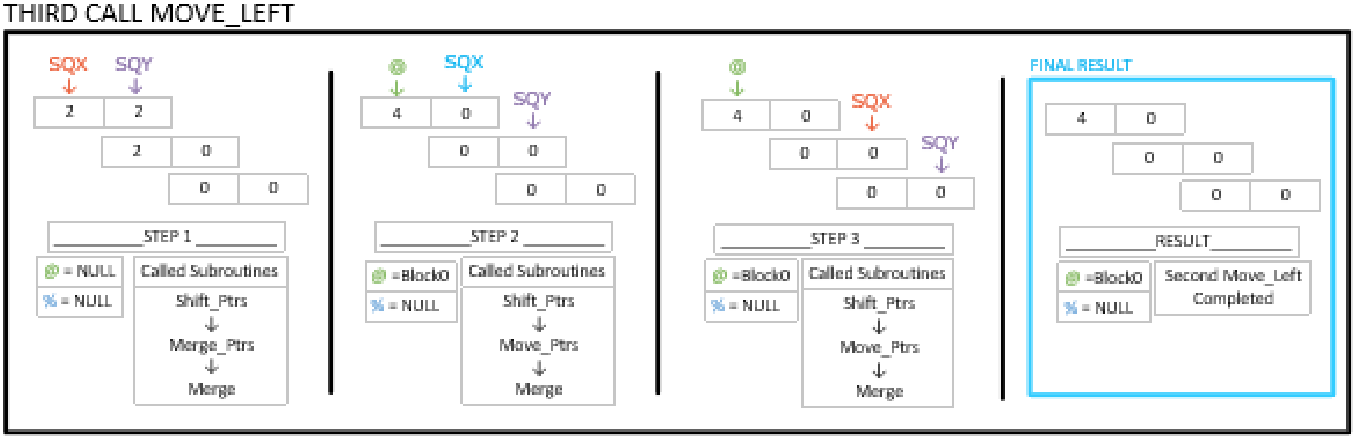
FIRST CALL MOVE_LEFT



SECOND CALL MOVE_LEFT



THIRD CALL MOVE_LEFT



Example 2

Single Capped Merge

2	**2**	**0**	**4**
Block 0	Block 1	Block 2	Block 3

Definition: Capped Merge, a block that has already been merged, and should be stopped from merging further

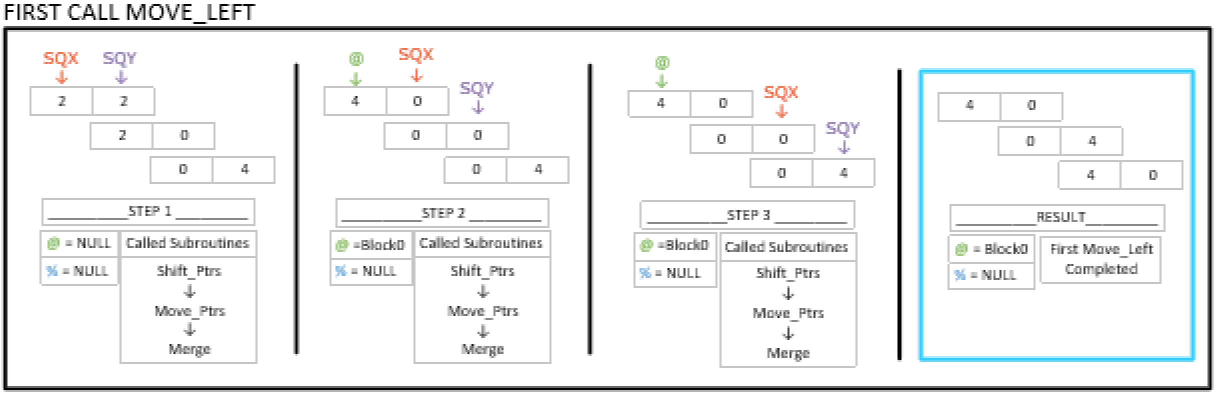
Explanation:

This example demonstrates the concept of a capped merge. The position of the capped merge is held in the pointers @ or %. The @ and % represent the (mergeA-mergeH) pointers in the code's comments.

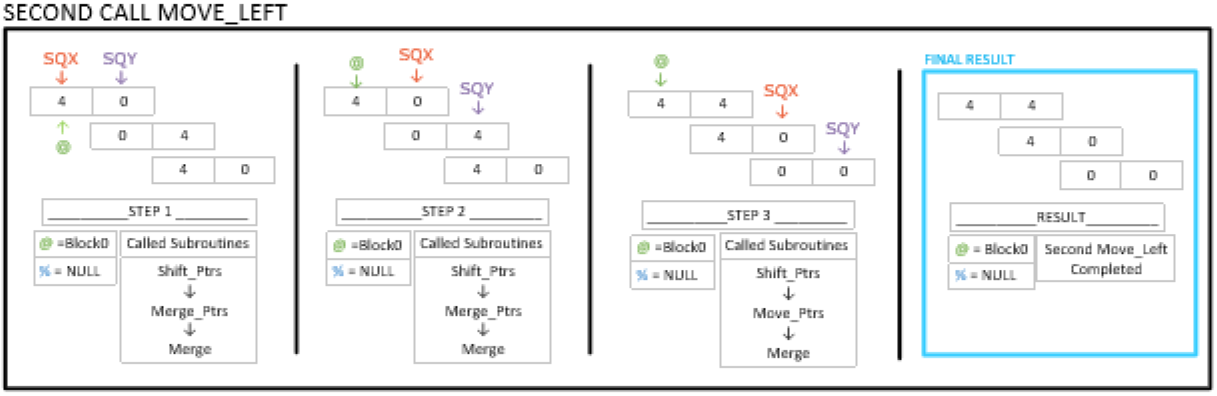
The @ or % are not cleared between calls of the movement function, allowing a capped merge that occurred in the first iteration to be recognized in the subsequent iterations.

Implementing the capped merge concept, we were able to prevent merging blocks that have already been merged.

FIRST CALL MOVE_LEFT



SECOND CALL MOVE_LEFT



SUBSEQUEINT THIRD CALL TO MOVE_LEFT WILL NOT CHANGE ANYTHING INVOLVING PTRS . . .

Example 3

Full Board (Double Merges)

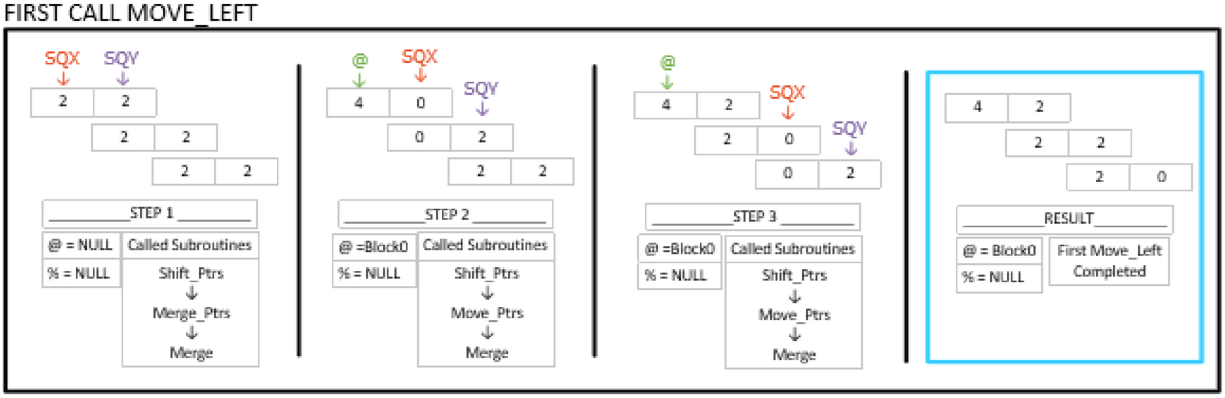
2	2	2	2
Block 0	Block 1	Block 2	Block 3

Definition: Double Merges, a double merge is when two capped merges occur on one row of the board.

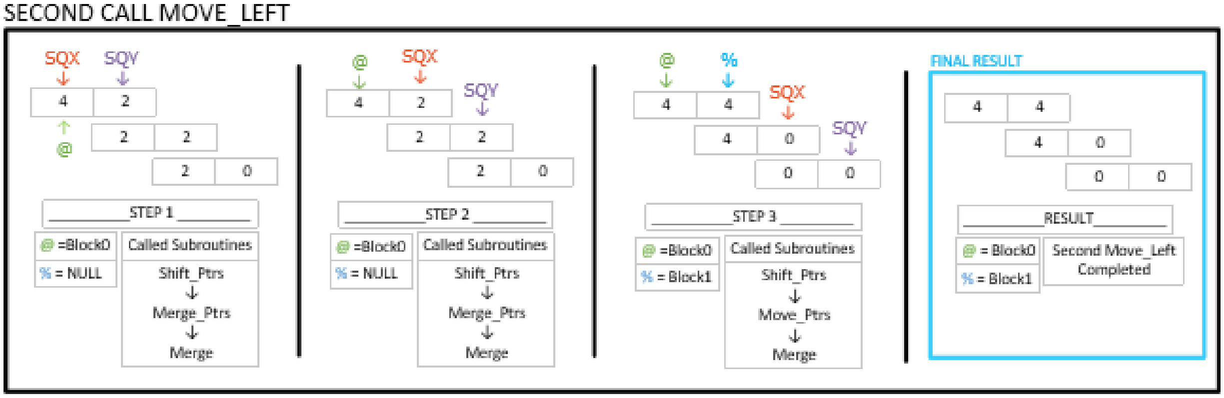
Explanation:

This example demonstrates why we decided to include both @ & % pointers in our implementation. Since there can occur a point at which more than one capped merge appears. In this situation two capped merges appear and are dealt with using both @ & % abstractions.

FIRST CALL MOVE_LEFT



SECOND CALL MOVE_LEFT



SUBSEQUEINT THIRD CALL TO MOVE_LEFT WILL NOT CHANGE ANYTHING INVOLVING PTRS . . .

Section 3:

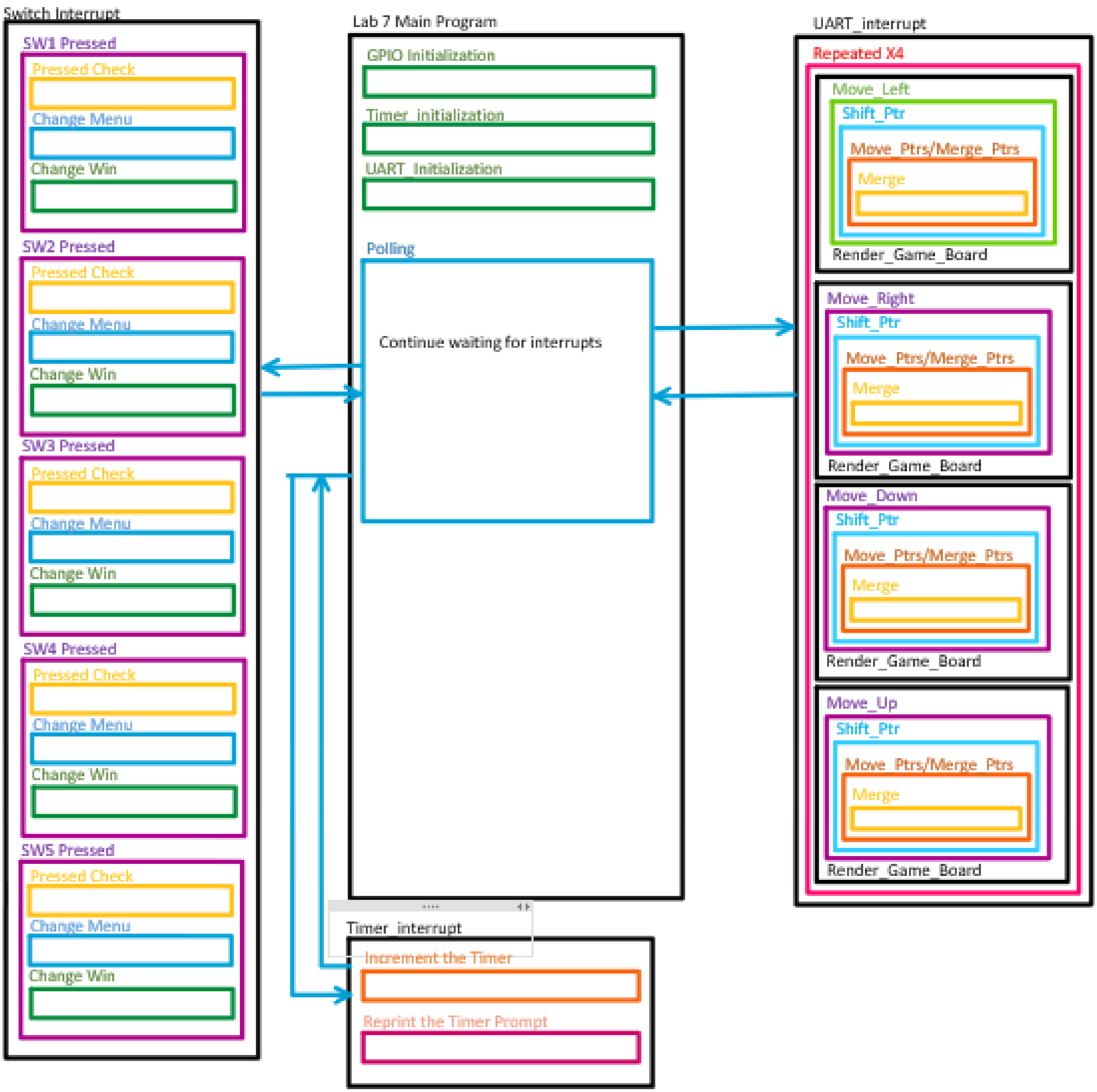
Program Overview:

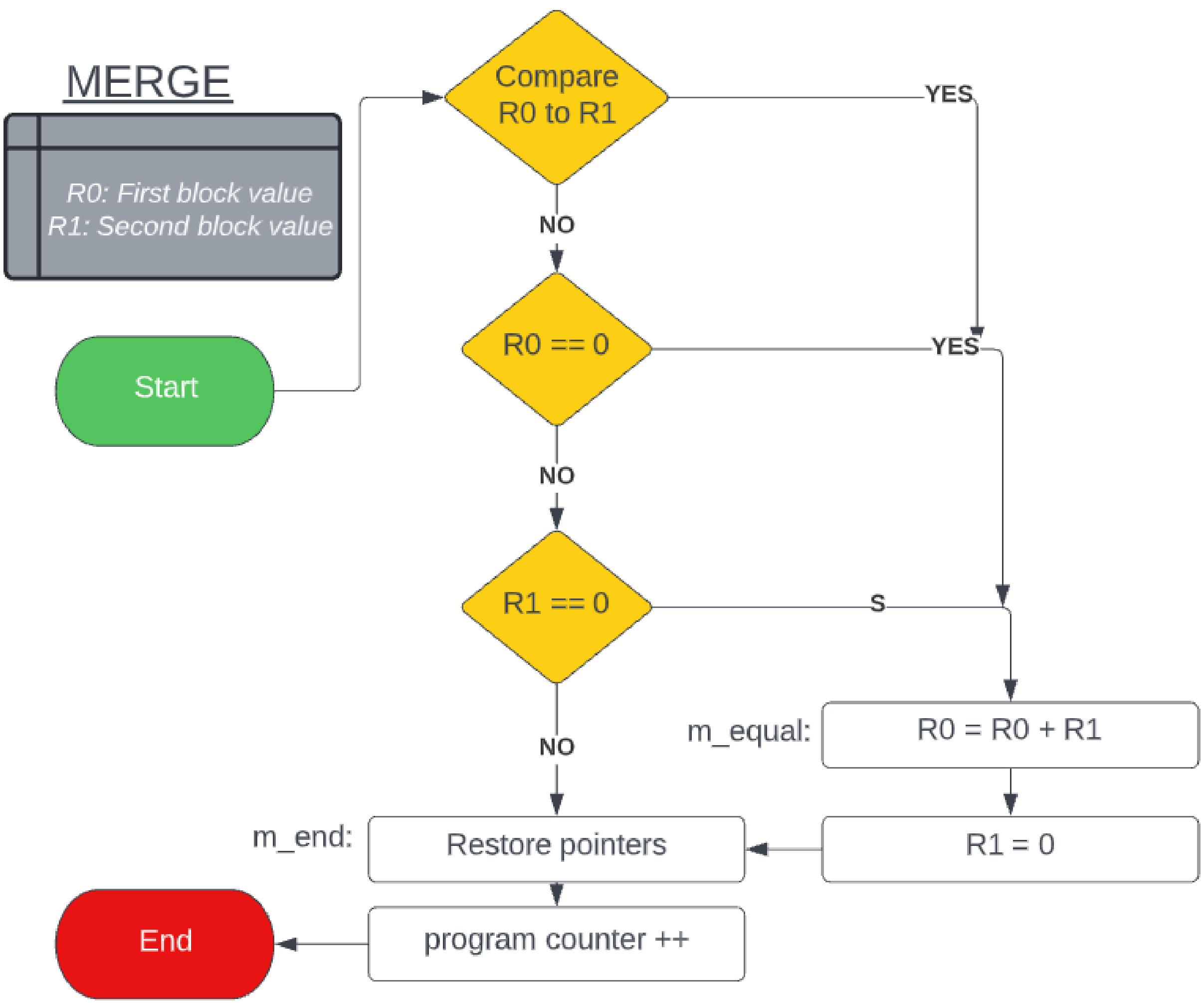
1. Start running the program, Start menu should be displayed in puTTy
   - Navigate the screen menus using the SW1-5 switches
2. The game board score prompt and 2 blocks should be displayed in puTTy
   - Using the “A”, “W”, “S”, “D” keys on the keyboard to shift the blocks in the desired direction
3. After each shift any matching blocks that were moved into each other will merge
   - When blocks are merged, the score will increment by the merged blocks’ value
4. While playing the game, the user can press the onboard Tiva switch to enter the pause menus. From there the user can quit the game, restart the current game, or change the win block condition.
   - When the block condition changes, the color on the Tiva will update to reflect the win condition based, documented LED color chart.
   - SW1 - Pause Game
   - SW2 - Quit game if paused, or change win block to 2048.
   - SW3 - Restart game if paused, or change win block to 1024
   - SW4 - Resume game, or change win block to 512
   - SW5 - Enter the change win block menu, or change win block to 256.
5. After the return from an interrupt the program will re-enter the infinite loop in the lab7 main subroutine.
6. This game will run indefinitely until the Tiva board itself is powered off.

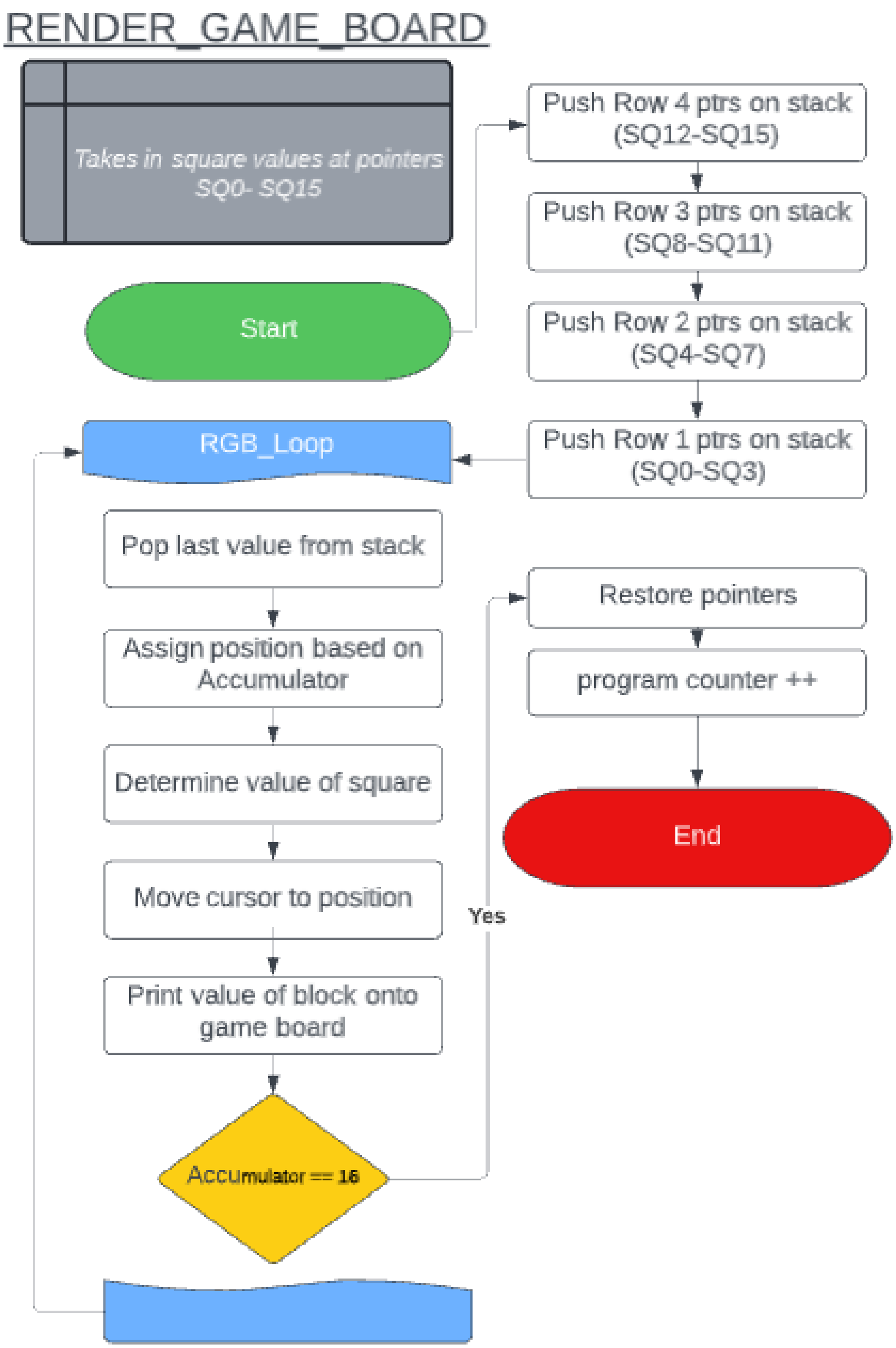
Program Summary:

In our implementation of the 2048 game, we utilize interrupt-based game mechanics for game input. A consequence of our design is the increase in interrupt latency as all of the game logic is performed within the UART handler. However, this reduces the number of redundant gameboard renderings significantly. Each pair of columns or rows, respectively, are rendered piecewise resulting in a smooth sliding animation for the blocks on each move operation.

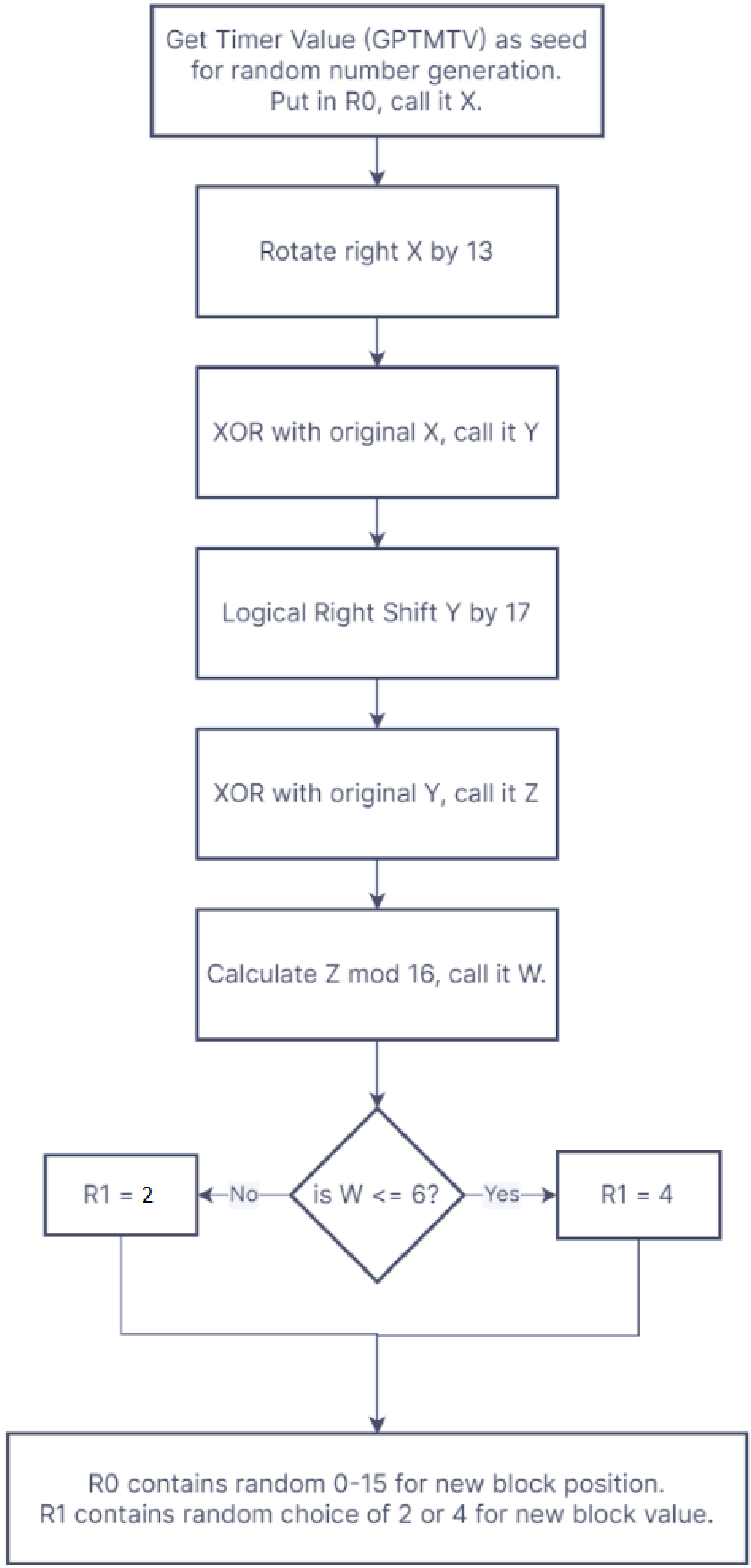
Section 4: *Subroutine flowcharts*

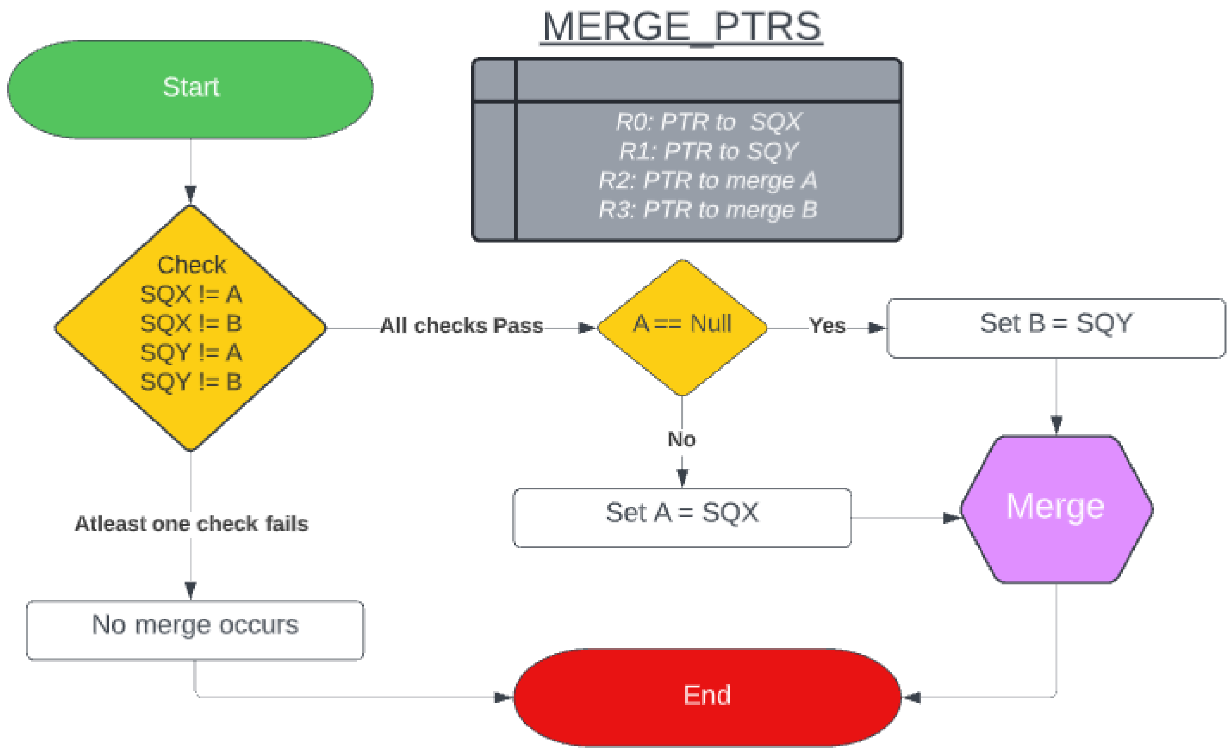


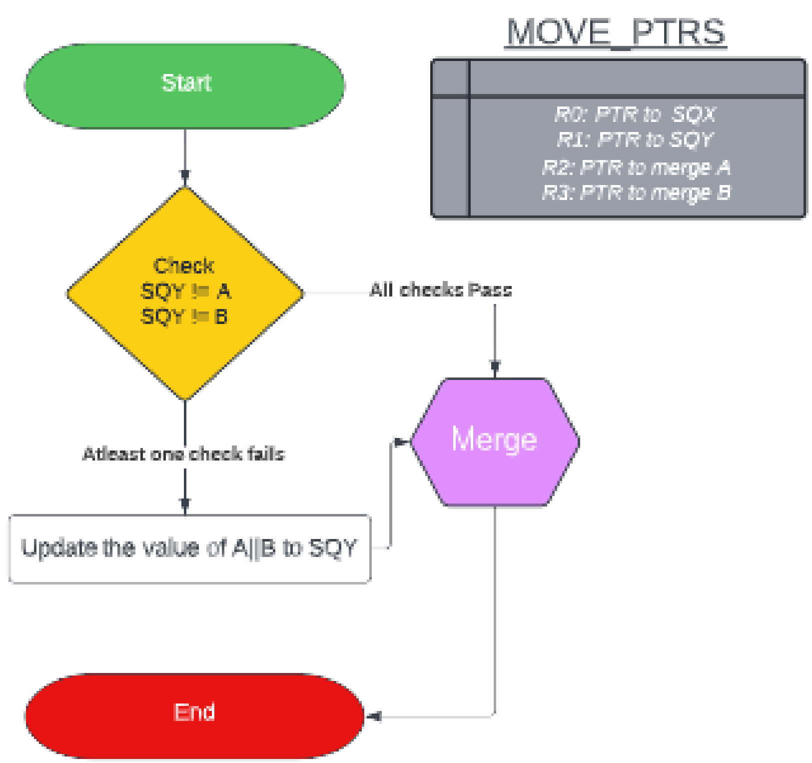

MERGE
R0: First block value
R1: Second block value
Start
Compare
R0 to R1
YES
NO
R0 == 0
YES
NO
R1 == 0
S
NO
m_equal:
R0 = R0 + R1
R1 = 0
m_end:
Restore pointers
program counter ++
End

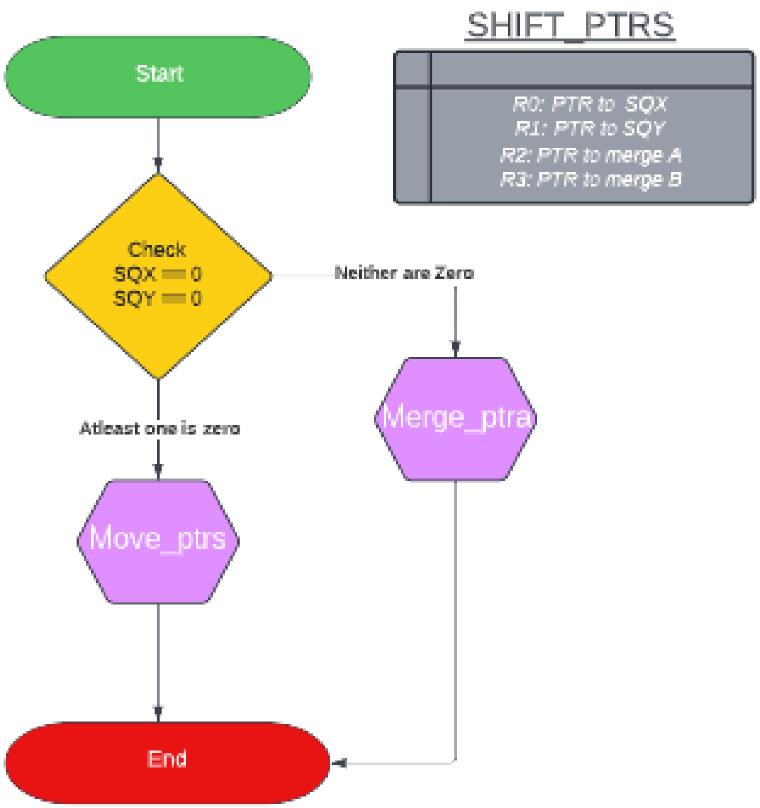

RENDER_GAME_BOARD
Takes in square values at pointers SQ0- SQ15
Start
Push Row 4 ptrs on stack (SQ12-SQ15)
Push Row 3 ptrs on stack (SQ8-SQ11)
Push Row 2 ptrs on stack (SQ4-SQ7)
Push Row 1 ptrs on stack (SQ0-SQ3)
RGB_Loop
Pop last value from stack
Assign position based on Accumulator
Determine value of square
Move cursor to position
Print value of block onto game board
Accumulator == 16
Yes
Restore pointers
program counter ++
End

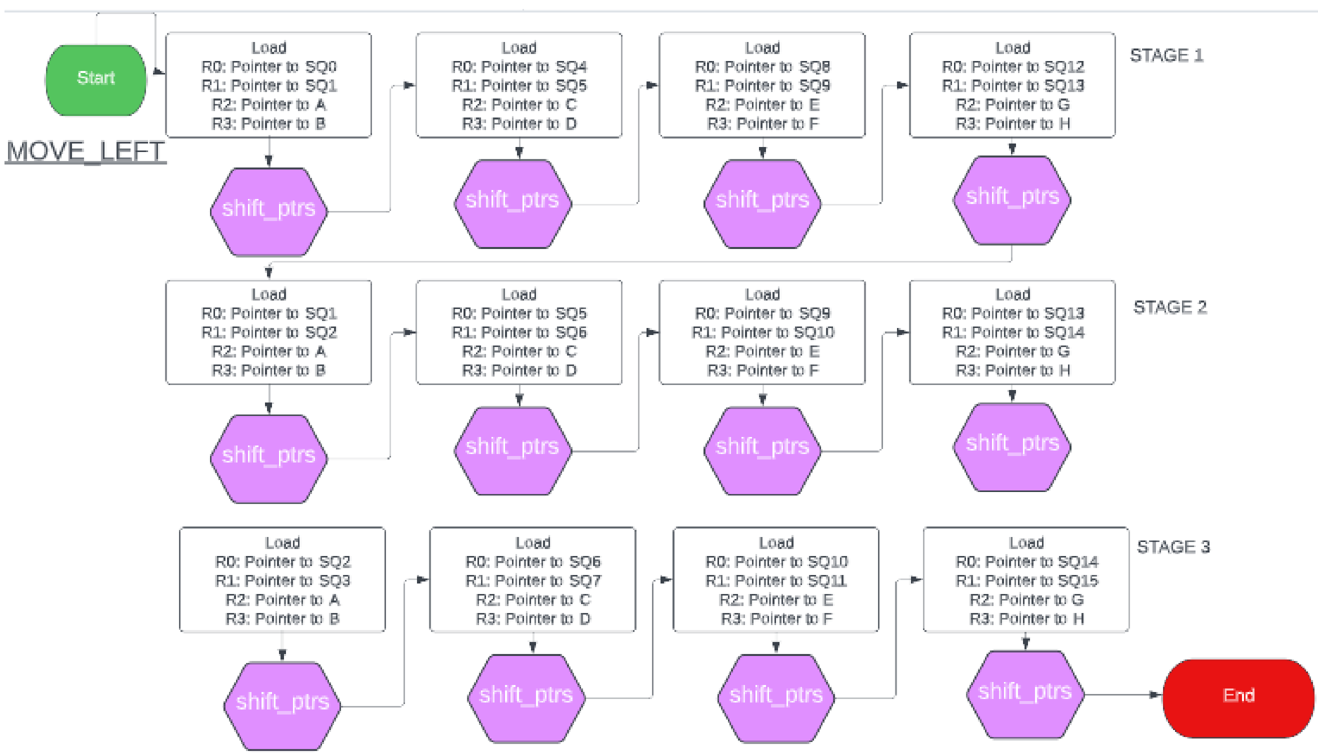
Random Number Generator

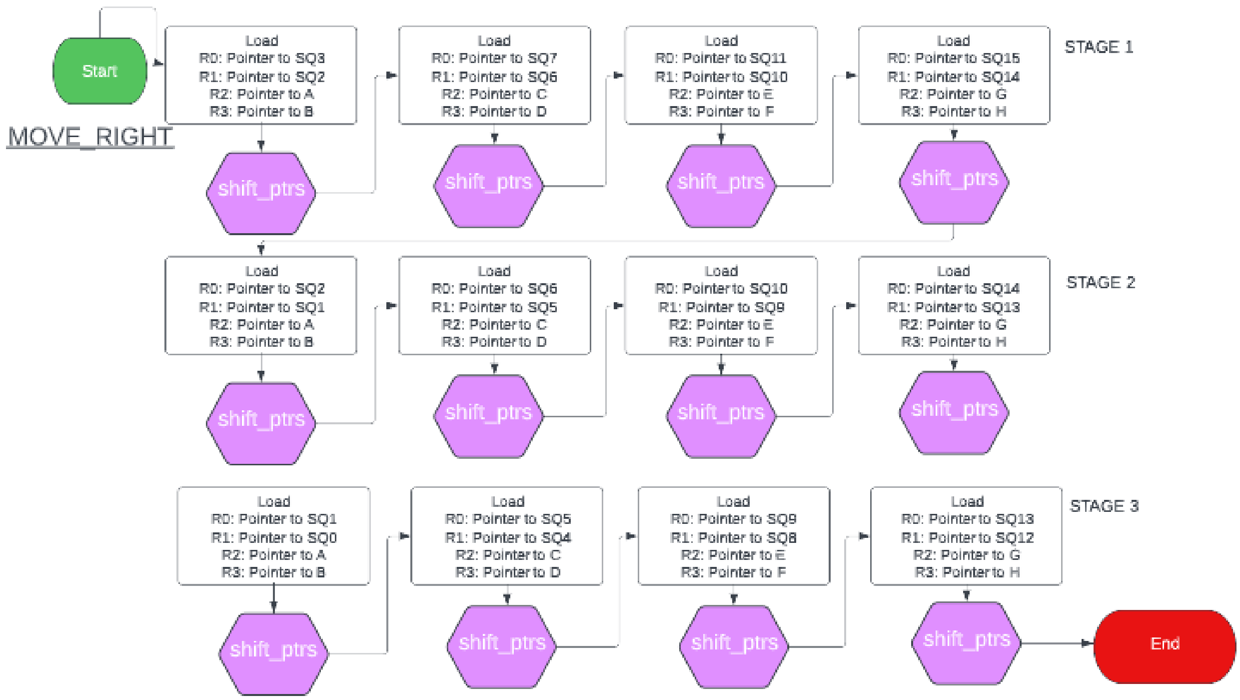


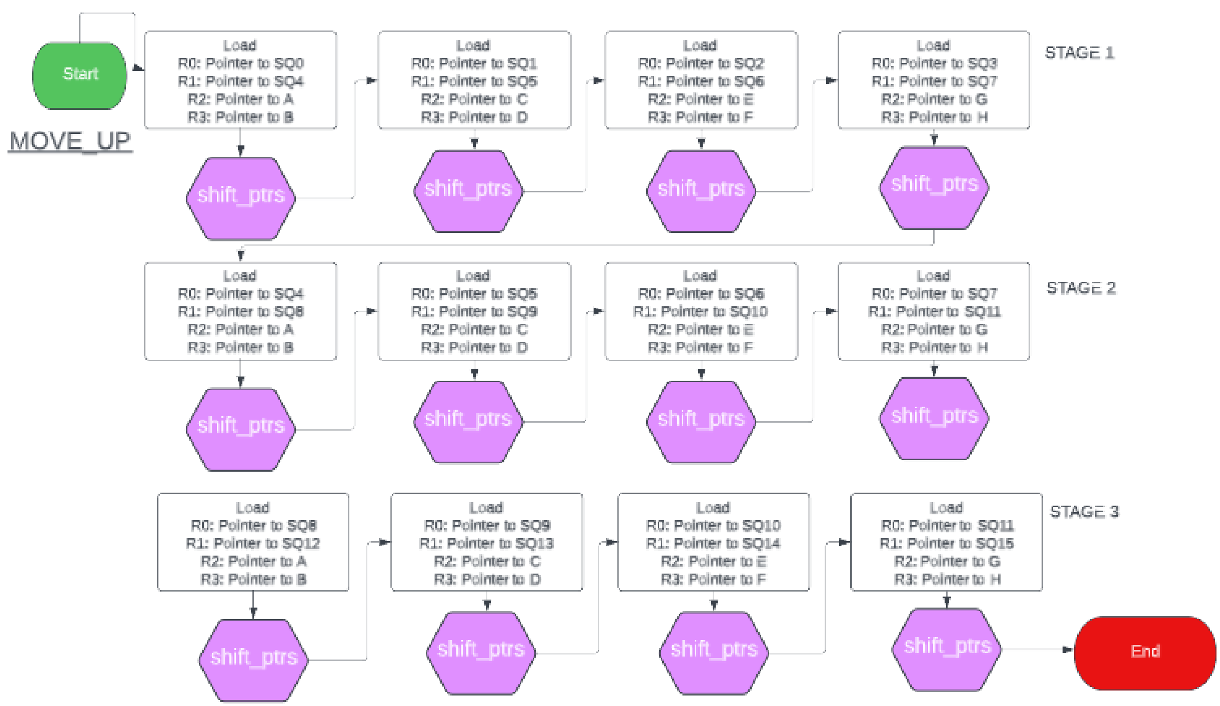

MERGE_PTRS
R0: PTR to SQX
R1: PTR to SQY
R2: PTR to merge A
R3: PTR to merge B
Start
Check
SQX != A
SQX != B
SQY != A
SQY != B
All checks Pass
A == Null
Yes
Set B = SQY
No
Set A = SQX
Merge
Atleast one check fails
No merge occurs
End

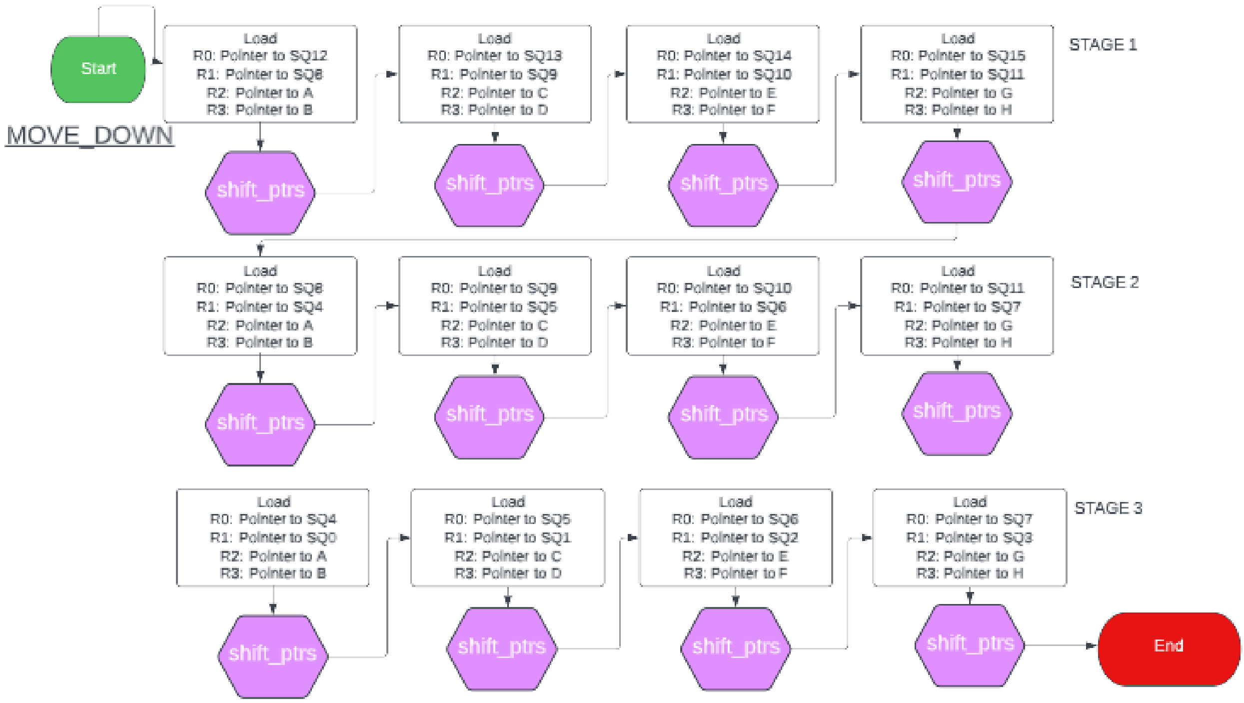

MOVE_PTRS
R0: PTR to SQX
R1: PTR to SQY
R2: PTR to merge A
R3: PTR to merge B
Start
Check
SQY != A
SQY != B
All checks Pass
Merge
Atleast one check fails
Update the value of A||B to SQY
End


SHIFT_PTRS
R0: PTR to SQX
R1: PTR to SQY
R2: PTR to merge A
R3: PTR to merge B
Start
Check
SQX == 0
SQY == 0
Neither are Zero
Merge_ptrs
Atleast one is zero
Move_ptrs
End

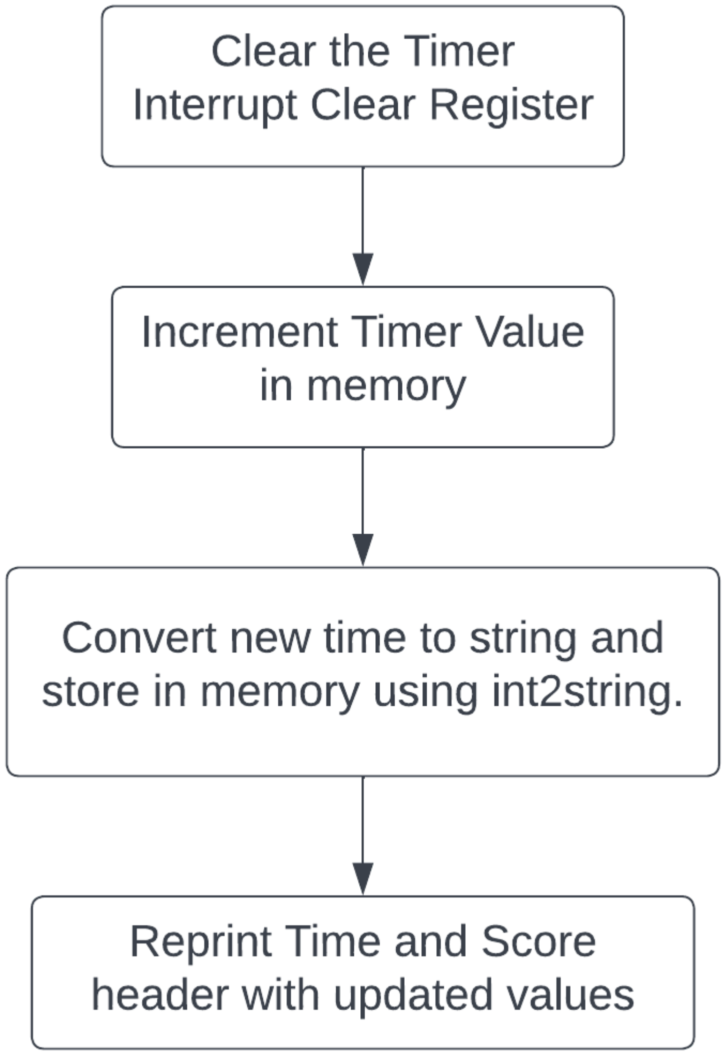

Start
MOVE_LEFT
Load
R0: Pointer to SQ0
R1: Pointer to SQ1
R2: Pointer to A
R3: Pointer to B
shift_ptrs
Load
R0: Pointer to SQ4
R1: Pointer to SQ5
R2: Pointer to C
R3: Pointer to D
shift_ptrs
Load
R0: Pointer to SQ8
R1: Pointer to SQ9
R2: Pointer to E
R3: Pointer to F
shift_ptrs
Load
R0: Pointer to SQ12
R1: Pointer to SQ13
R2: Pointer to G
R3: Pointer to H
shift_ptrs
STAGE 1
Load
R0: Pointer to SQ1
R1: Pointer to SQ2
R2: Pointer to A
R3: Pointer to B
shift_ptrs
Load
R0: Pointer to SQ5
R1: Pointer to SQ6
R2: Pointer to C
R3: Pointer to D
shift_ptrs
Load
R0: Pointer to SQ9
R1: Pointer to SQ10
R2: Pointer to E
R3: Pointer to F
shift_ptrs
Load
R0: Pointer to SQ13
R1: Pointer to SQ14
R2: Pointer to G
R3: Pointer to H
shift_ptrs
STAGE 2
Load
R0: Pointer to SQ2
R1: Pointer to SQ3
R2: Pointer to A
R3: Pointer to B
shift_ptrs
Load
R0: Pointer to SQ6
R1: Pointer to SQ7
R2: Pointer to C
R3: Pointer to D
shift_ptrs
Load
R0: Pointer to SQ10
R1: Pointer to SQ11
R2: Pointer to E
R3: Pointer to F
shift_ptrs
Load
R0: Pointer to SQ14
R1: Pointer to SQ15
R2: Pointer to G
R3: Pointer to H
shift_ptrs
STAGE 3
End


Start
MOVE_RIGHT
Load
R0: Pointer to SQ3
R1: Pointer to SQ2
R2: Pointer to A
R3: Pointer to B
shift_ptrs
Load
R0: Pointer to SQ7
R1: Pointer to SQ6
R2: Pointer to C
R3: Pointer to D
shift_ptrs
Load
R0: Pointer to SQ11
R1: Pointer to SQ10
R2: Pointer to E
R3: Pointer to F
shift_ptrs
Load
R0: Pointer to SQ15
R1: Pointer to SQ14
R2: Pointer to G
R3: Pointer to H
shift_ptrs
STAGE 1
Load
R0: Pointer to SQ2
R1: Pointer to SQ1
R2: Pointer to A
R3: Pointer to B
shift_ptrs
Load
R0: Pointer to SQ6
R1: Pointer to SQ5
R2: Pointer to C
R3: Pointer to D
shift_ptrs
Load
R0: Pointer to SQ10
R1: Pointer to SQ9
R2: Pointer to E
R3: Pointer to F
shift_ptrs
Load
R0: Pointer to SQ14
R1: Pointer to SQ13
R2: Pointer to G
R3: Pointer to H
shift_ptrs
STAGE 2
Load
R0: Pointer to SQ1
R1: Pointer to SQ0
R2: Pointer to A
R3: Pointer to B
shift_ptrs
Load
R0: Pointer to SQ5
R1: Pointer to SQ4
R2: Pointer to C
R3: Pointer to D
shift_ptrs
Load
R0: Pointer to SQ9
R1: Pointer to SQ8
R2: Pointer to E
R3: Pointer to F
shift_ptrs
Load
R0: Pointer to SQ13
R1: Pointer to SQ12
R2: Pointer to G
R3: Pointer to H
shift_ptrs
STAGE 3
End

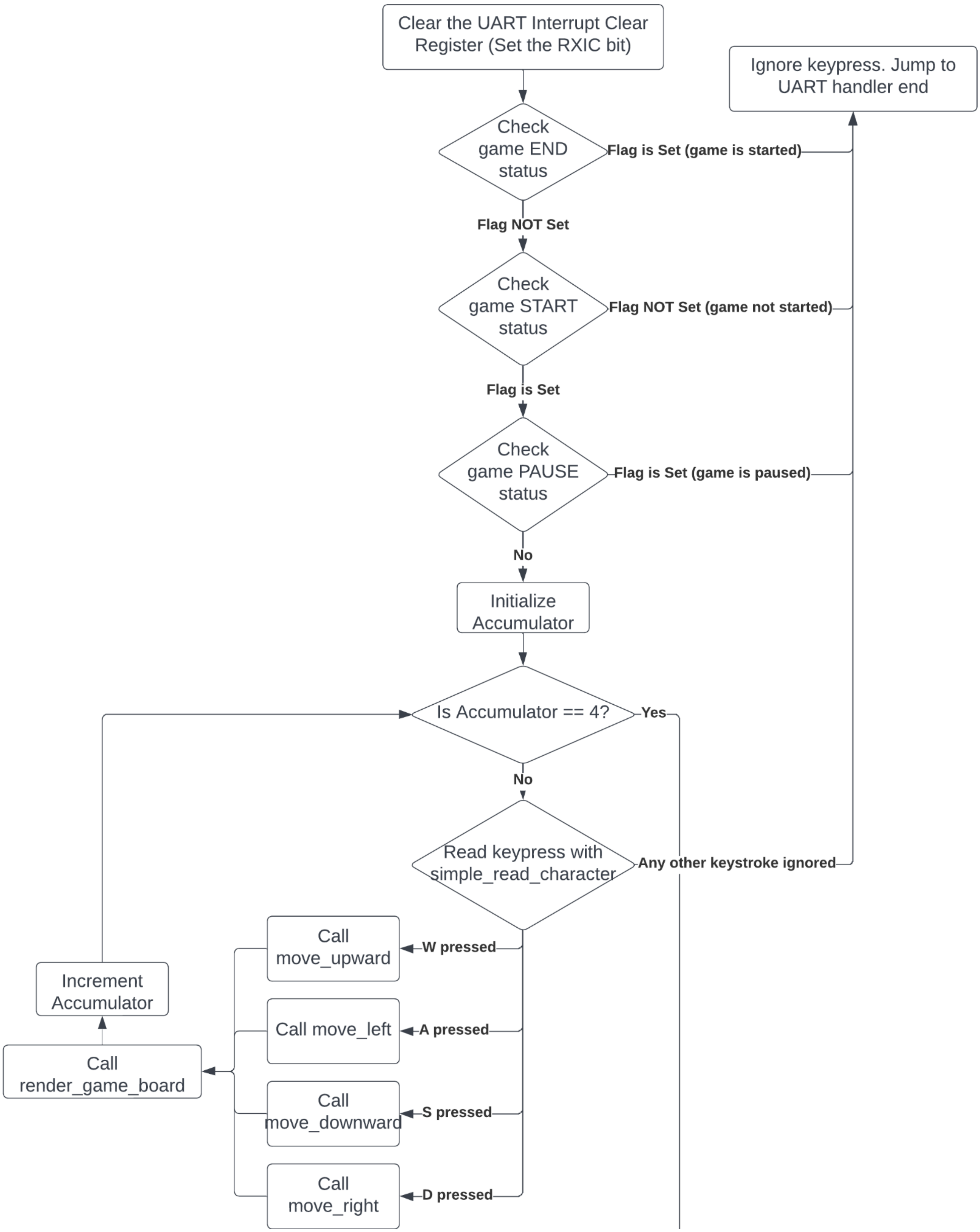

Start
MOVE_UP
Load
R0: Pointer to SQ0
R1: Pointer to SQ4
R2: Pointer to A
R3: Pointer to B
shift_ptrs
Load
R0: Pointer to SQ1
R1: Pointer to SQ5
R2: Pointer to C
R3: Pointer to D
shift_ptrs
Load
R0: Pointer to SQ2
R1: Pointer to SQ6
R2: Pointer to E
R3: Pointer to F
shift_ptrs
Load
R0: Pointer to SQ3
R1: Pointer to SQ7
R2: Pointer to G
R3: Pointer to H
shift_ptrs
STAGE 1
Load
R0: Pointer to SQ4
R1: Pointer to SQ8
R2: Pointer to A
R3: Pointer to B
shift_ptrs
Load
R0: Pointer to SQ5
R1: Pointer to SQ9
R2: Pointer to C
R3: Pointer to D
shift_ptrs
Load
R0: Pointer to SQ6
R1: Pointer to SQ10
R2: Pointer to E
R3: Pointer to F
shift_ptrs
Load
R0: Pointer to SQ7
R1: Pointer to SQ11
R2: Pointer to G
R3: Pointer to H
shift_ptrs
STAGE 2
Load
R0: Pointer to SQ8
R1: Pointer to SQ12
R2: Pointer to A
R3: Pointer to B
shift_ptrs
Load
R0: Pointer to SQ9
R1: Pointer to SQ13
R2: Pointer to C
R3: Pointer to D
shift_ptrs
Load
R0: Pointer to SQ10
R1: Pointer to SQ14
R2: Pointer to E
R3: Pointer to F
shift_ptrs
Load
R0: Pointer to SQ11
R1: Pointer to SQ15
R2: Pointer to G
R3: Pointer to H
shift_ptrs
STAGE 3
End


MOVE_DOWN
Start
Load
R0: Pointer to SQ12
R1: Pointer to SQ8
R2: Pointer to A
R3: Pointer to B
shift_ptrs
Load
R0: Pointer to SQ13
R1: Pointer to SQ9
R2: Pointer to C
R3: Pointer to D
shift_ptrs
Load
R0: Pointer to SQ14
R1: Pointer to SQ10
R2: Pointer to E
R3: Pointer to F
shift_ptrs
Load
R0: Pointer to SQ15
R1: Pointer to SQ11
R2: Pointer to G
R3: Pointer to H
shift_ptrs
STAGE 1
Load
R0: Pointer to SQ8
R1: Pointer to SQ4
R2: Pointer to A
R3: Pointer to B
shift_ptrs
Load
R0: Pointer to SQ9
R1: Pointer to SQ5
R2: Pointer to C
R3: Pointer to D
shift_ptrs
Load
R0: Pointer to SQ10
R1: Pointer to SQ6
R2: Pointer to E
R3: Pointer to F
shift_ptrs
Load
R0: Pointer to SQ11
R1: Pointer to SQ7
R2: Pointer to G
R3: Pointer to H
shift_ptrs
STAGE 2
Load
R0: Pointer to SQ4
R1: Pointer to SQ0
R2: Pointer to A
R3: Pointer to B
shift_ptrs
Load
R0: Pointer to SQ5
R1: Pointer to SQ1
R2: Pointer to C
R3: Pointer to D
shift_ptrs
Load
R0: Pointer to SQ6
R1: Pointer to SQ2
R2: Pointer to E
R3: Pointer to F
shift_ptrs
Load
R0: Pointer to SQ7
R1: Pointer to SQ3
R2: Pointer to G
R3: Pointer to H
shift_ptrs
STAGE 3
End

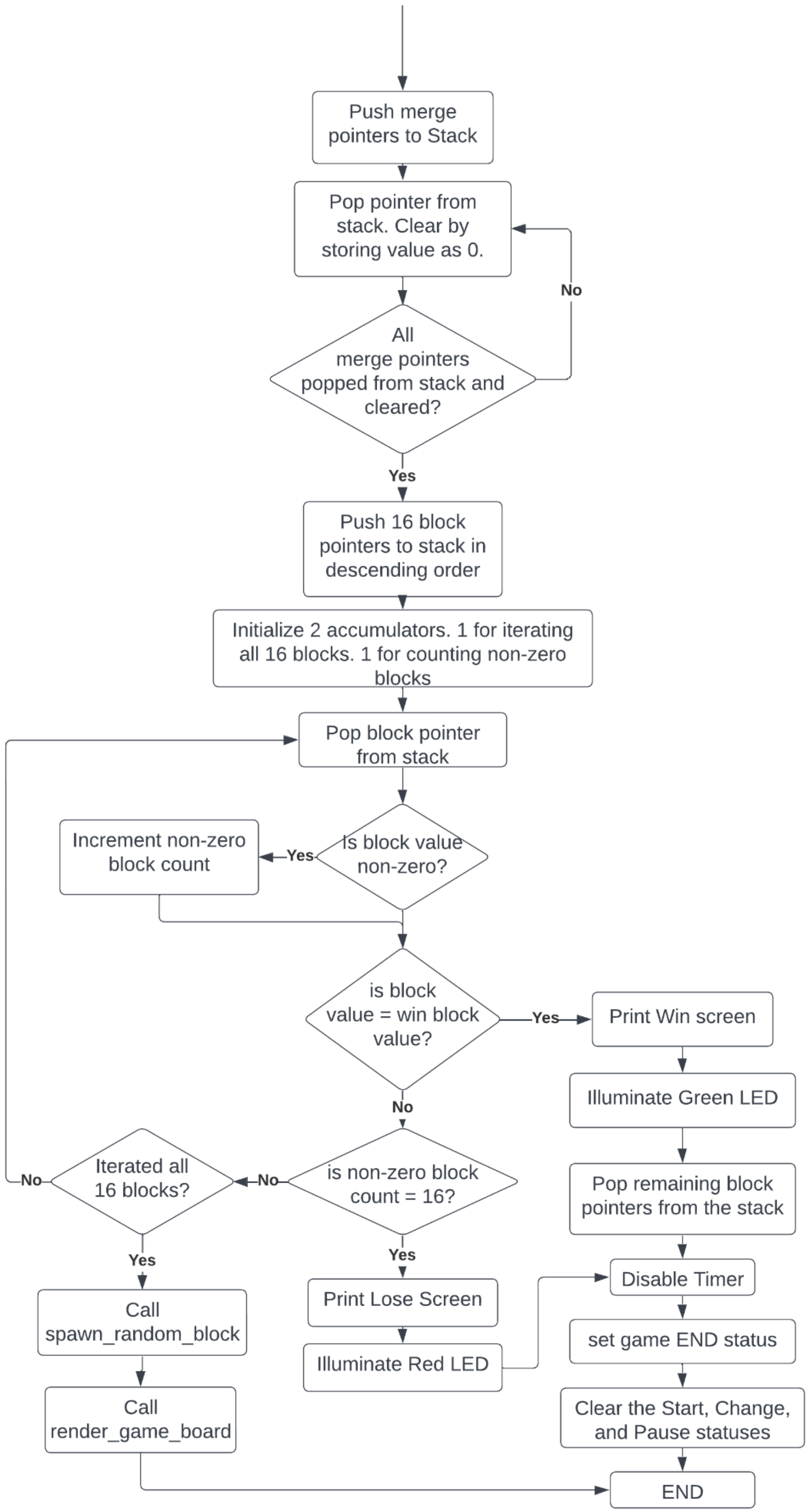
Timer Interrupt Handler



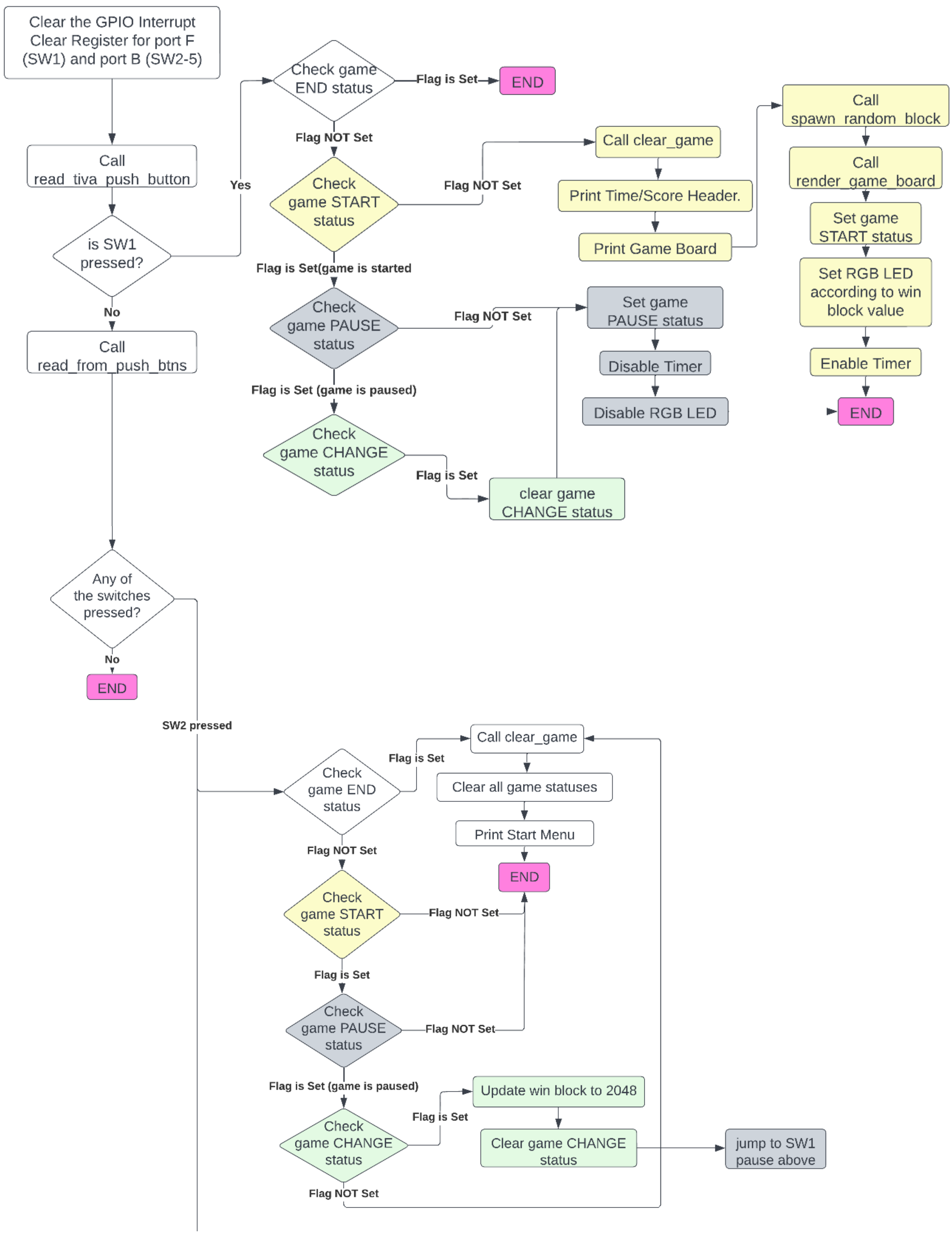
UART Interrupt Handler

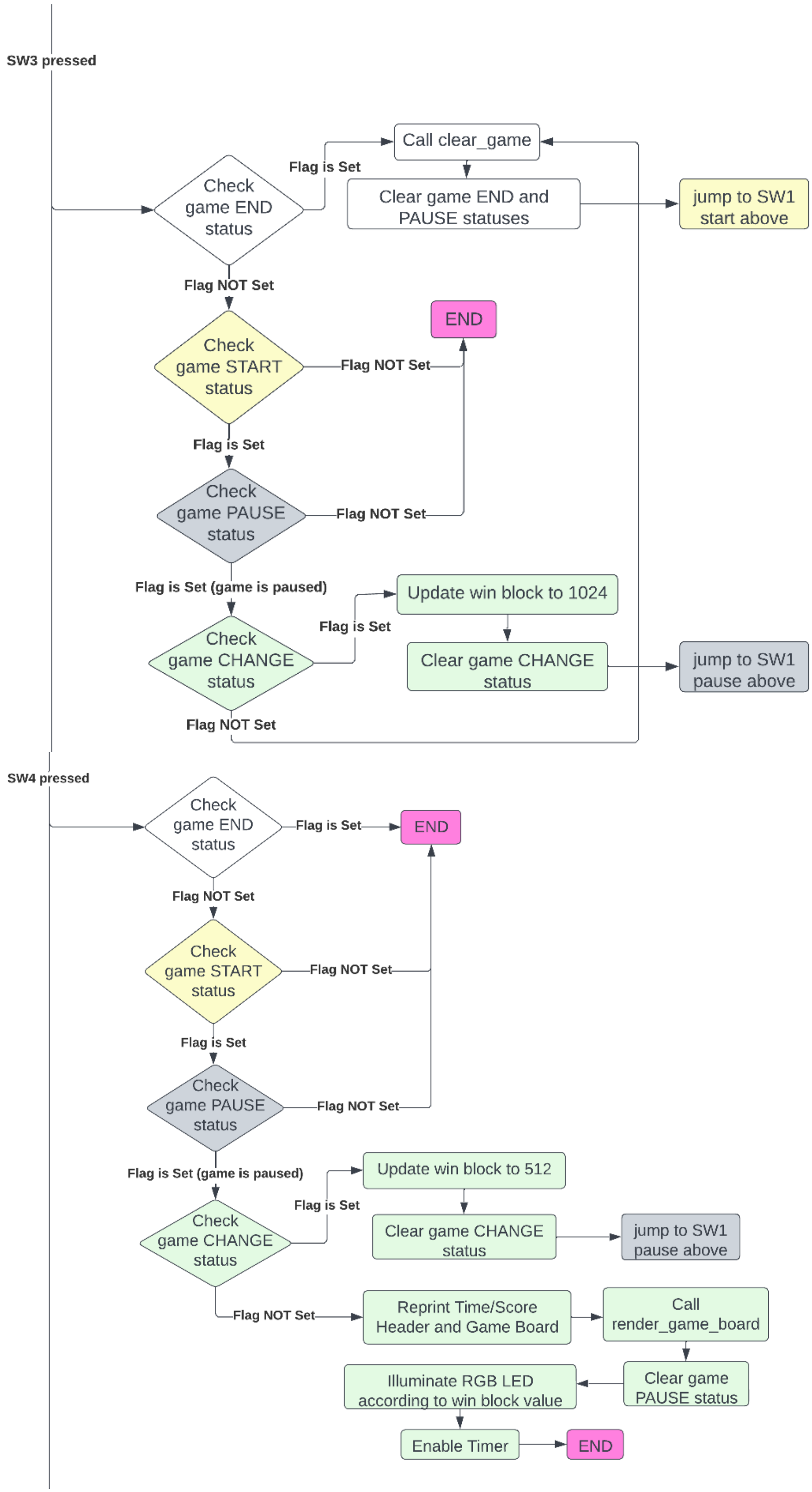


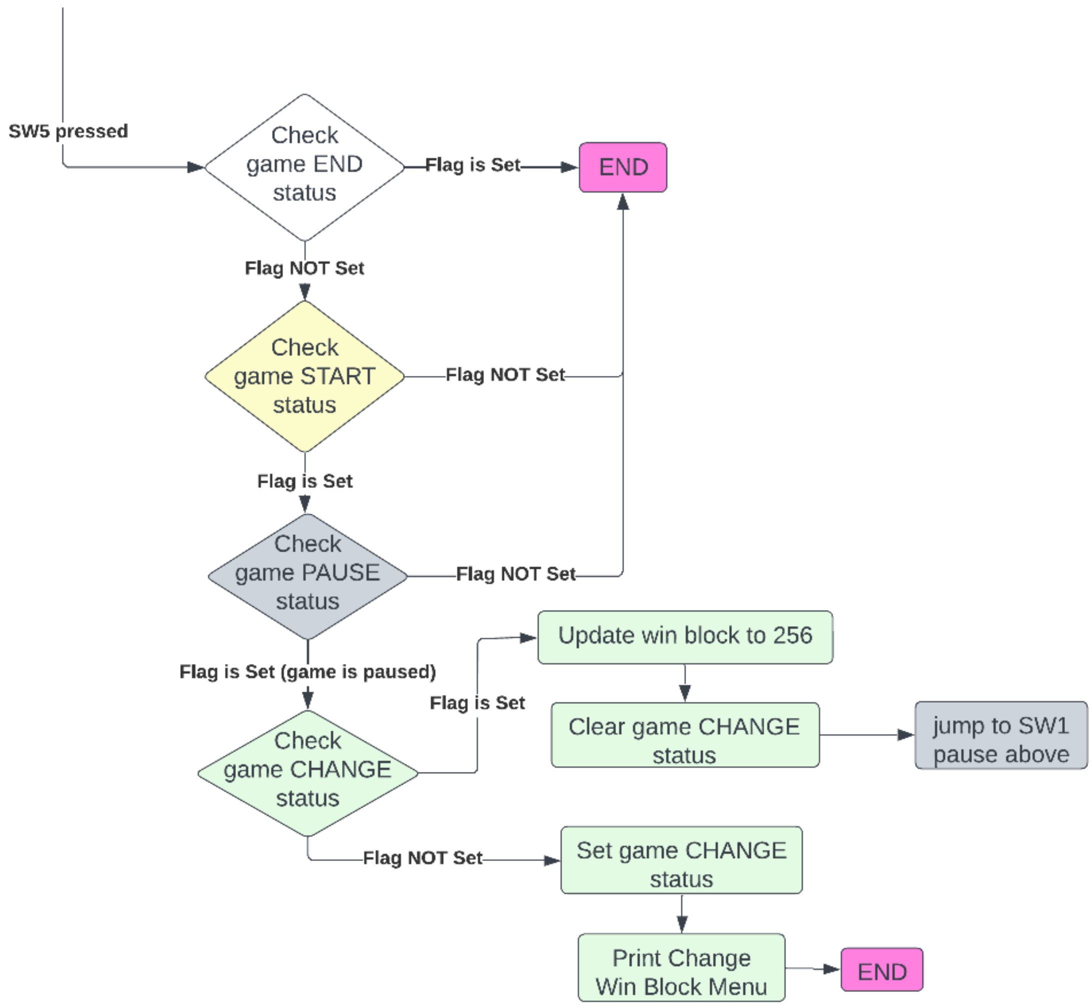
UART Interrupt Handler cont.



Switch Interrupt Handler




SW3 pressed
Check game END status
Flag is Set
Call clear_game
Clear game END and PAUSE statuses
jump to SW1 start above
Flag NOT Set
Check game START status
Flag NOT Set
END
Flag is Set
Check game PAUSE status
Flag NOT Set
Flag is Set (game is paused)
Check game CHANGE status
Flag is Set
Update win block to 1024
Clear game CHANGE status
jump to SW1 pause above
Flag NOT Set
SW4 pressed
Check game END status
Flag is Set
END
Flag NOT Set
Check game START status
Flag NOT Set
Flag is Set
Check game PAUSE status
Flag NOT Set
Flag is Set (game is paused)
Check game CHANGE status
Flag is Set
Update win block to 512
Clear game CHANGE status
jump to SW1 pause above
Flag NOT Set
Reprint Time/Score Header and Game Board
Call render_game_board
Clear game PAUSE status
Illuminate RGB LED according to win block value
Enable Timer
END


SW5 pressed
Check game END status
Flag is Set
END
Flag NOT Set
Check game START status
Flag NOT Set
Flag is Set
Check game PAUSE status
Flag NOT Set
Flag is Set (game is paused)
Update win block to 256
Flag is Set
Clear game CHANGE status
jump to SW1 pause above
Check game CHANGE status
Flag NOT Set
Set game CHANGE status
Print Change Win Block Menu
END