EE314 Digital Electronics Laboratory 2024-2025 Spring Term Project

Engin Ege, 2575181, Alkan Alper, 2574556, Büyük Necati, 2574838

Abstract—**In this report, we will explain both the theoretical and practical parts of the project, which is inspired by the 2D two-player fighting game called FOOTSIES. [2] This game was developed entirely by using Verilog HDL on the FPGA platform. With the help of this project, we will be gaining experience in digital design fundamentals such as real-time graphics rendering, finite state machines (FSMs), and input debouncing. Hence, the project emphasizes core digital design concepts through interactive, hardware-based games.**

Keywords—Digital Design, Verilog HDL, Logic Design, 2D two-player fighting game, FSM, Real-time graphics rendering.

I. Introduction

In this project, we will implement a two-player game on an FPGA using only Verilog HDL to implement the digital design concepts. The gameplay includes character movement, basic/directional attacks, and blocking. The game that we designed provides two game modes, such as human versus human and human versus bot, displayed via a VGA interface, and the characters were controlled by physical keypads. The game logic is controlled by using finite state machines (FSMs) to manage the character behavior and game flow. Key features in the design include a menu selection screen, a countdown animation, gameplay logic displayed on VGA at 60Hz, and hit detection through a hitscan module. In this report, we will be explaining each key feature explained above.

II. Game Start and Initialization

In this section, the game will start from the game menu, which consists of 2 game modes, such as human versus human and human versus bot. After selecting the game mode and getting confirmation from the user, the countdown will start. This will be followed by the transition to the gameplay. We will be handling these by using two modules written in Verilog, which are game_menu and countdown.

A. Game Menu Module

The game_menu module handles the display and the interaction with the user to select the game mode. Then make the correct transition to the countdown state.

A.1 Inputs and Outputs of the Game Menu Module: This module will take inputs from the user by toggling the switch between the game modes (1 player vs. bot or 2 players). Other input is taken from the confirm button from the user to start the countdown. The outputs for this module are to color the corresponding pixels on the screen, which is the pixel_color output, to trigger the countdown module, which is the trigger_count_start output, and two 7-segment display outputs, which are hexout and hexout2, to show the selected game mode.

A.2 Display Logic for Game Menu Module: In this module, we will be displaying three text items: “Menu”, “P1”, and “P2”. The location of the “Menu” is fixed on the screen since it is the title. The switch selects between game modes when it is high; the “P2” option is highlighted on the screen. When it is low, the “P1” option is highlighted. The background color is selected as white.

A.3 Game Start Trigger for Game Menu Module: After selecting the game mode, the user will confirm his/her selection by pressing the confirm button. When the confirm button is pressed, the trigger_count_start signal will start the countdown phase, and simultaneously, the 7-segment displays the selection of the user (i.e., P2 for a 2-player game or P1 for a 1-player game).

B.Countdown Module

The countdown module displays a sequence such as “3-2-1-START” on the screen before the game starts.

B.1 Inputs and Outputs: For the inputs, we have two signals for the clock which are the main clock signal clk, and the clk60 for control of the timing, reset_n for active low reset for the countdown, game_state which indicates the game state, and next_x and next_y which indicate the coordinates of the texts on the VGA screen.

B.2 FSM for Countdown Module: We used FSM logic to handle the countdown loop. We have four states: {S_THREE, S_TWO, S_ONE, S_START}. We controlled the duration in the first three states by counting the 60 frames of the 60Hz clock, which lasts approximately 1 second. When the countdown is finished, FSM goes to the S_START state, which triggers the trigger_game_start signal to raise the flag to transition to the gameplay state. The state diagram for the FSM of the Countdown Module can be found in Appendix A.

B.3 Display Logic for Countdown Module: At each state, “3”, “2”, “1”, and “START” will be displayed respectively. Each state will last 1 second to have a smooth sequence.

III. Gameplay Loop

In the gameplay loop, we had mainly 4 modules to determine the game logic. These are the FSM, Game1P, Game2P, Hitscan, and PlayerMovement modules.

A.FSM Module

In this module, we encoded the character’s status as idle, forward, backward, basic attack start, basic attack active, basic attack recovery, basic attack start, basic attack active, basic attack recovery, hit stun, and block stun states.

A.1 Inputs of the FSM Module: We had a 1-bit clk input where the state transitions and frame counting occur. Then we had a 1-bit reset_n input when the reset is high, all the states, counters, and registers are set to their initial values. We took the input from the user for character movement by using

3-bit buttons[2:0]. “Buttons [2]” is for moving backward, “buttons[1]” is for moving forward, and “buttons[0]” is for attacking. We have a 1-bit hitscan input, which indicates that the opponent’s attack hit this player on the previous frame. Finally, we have a 4-bit input named Opponent_CS, which is for the opponent’s current state. This is used to determine the time that the player will remain in specific stun states, such as basic and directional attacks.

A.2 Outputs of the FSM Module: We had a 4-bit output named P_Current_State, which shows the current state of the player, and other modules (movement, rendering, and hit detection) use this input to know what the player is doing. We had a 3-bit output named lives_led, which shows the remaining lives of the player, and we had a 3-bit output named blocks_led, which shows the remaining blocks of the player. Finally, we had a 1-bit died output, which is 1 when the lives reach zero. Otherwise, it is low.

A.3 Internal Registers and Wires of the FSM Module: We defined a 2-bit lives register for encoding remaining lives, and the same is done for the blocks register for encoding the remaining blocks. We defined a 5-bit frame counter register for counting up to the required frame for the corresponding state. On each 60Hz clock tick, it either increments by one or resets to zero when the state changes. A 4-bit register called State_Frame_Holder is defined based on the opponent’s attack (i.e., basic or directional) to temporarily hold the number of frames to remain in a state based on the description. A 4-bit register called NS is defined to represent the next state of the character, and it is used in the always block. Two 2-bit registers called life_control and block_control are defined for sending control signals to the shifter to either select “HOLD” or “SERIAL_L” (shift-left) operations.

A.4 Logic of the FSM Module: We decided to check the hit/block interruption at the top of the FSM logic. FSM checks whether hitscan is high or not, and the player is not already in the stun state or block state. If this is the case, it forces a transition into either state. A detailed state diagram for the FSM logic can be found in Appendix B. We will explain the states and the transitions from each state.

A.4.1 Block-Stun State: If the player’s current state is S_BACKWARD, the FSM chooses S_BLOCK_STUN, and then block_control is set to SERIAL_L to reduce the block count by one by shifting the block shift register. Then, State_Frame_Holder is loaded with 14 frames if the opponent’s current state is S_DIR_ATTACK_ACTIVE state or loaded with 15 frames if the opponent is in S_BASIC_ATTACK_ACTIVE state. The aim is to differentiate the durations for different attacks, such as basic and directional.

A.4.2 Hit-Stun State: If the player is not in the Block-Stun state, then FSM transitions to the S_HIT_STUN state. Then, since the player is not blocking any attack, it will get attacked and lose health. We decremented the health by setting the life_control to SERIAL_L so that the shift register shifted out 1 health bar. If “hitscan =0), then the player’s state transitions into a case called P_Current_State. With this large case block, we will handle every possible transition.

A.4.3 Idle, Forward, Backward States: The player starts from S_IDLE by default. If no button or “right” & “left” is pressed, the state remains as S_IDLE. If “attack & right & left” or only “attack” is pressed, the next state is S_BASIC_ATTACK_START. If “attack & right” or “attack & left” is pressed, the next state is S_DIR_ATTACK_START. If only “left” is pressed, the next state is S_BACKWARD. If only “right” is pressed, the next state is S_FORWARD.

A.4.4 Basic Attack Start/Active/Recovery States: For the character animations, the FSM will count the frames such that when an animation reaches its desired frame, it transitions into the other. When the character is in S_BASIC_ATTACK_START, it will remain in this state until frame_count reaches 5. Once the frame_count = 5, the next state is S_DIR_ATTACK_ACTIVE. In this state, the character will remain until the frame_count reaches 2. When frame_count reaches 2, the next state is S_BASIC_ATTACK_REC. In the recovery state, the FSM counts 16 frames. When it reaches 16, then just like in the idle state, the user can press any button combinations, and the FSM transitions to the states as in the idle state accordingly.

A.4.5 Directional Attack Start/Active/Recovery: The same logic that we used for basic attack start/active/recovery applies to this state also. The only difference is the difference in the number of frames required for the transitions between states. In the startup, the duration is shorter than the basic attack state, which is 4 frames. In the active state, the duration is 3 frames, and in the recovery, 15 frames for recovery in the directional attack. After the recovery, the user can press any button just like in the idle state and can transition to the same state as in the idle case.

A.4.6 Hit-Stun State: The character remains in hit-stun until frame_count == State_Frame_Holder. If the hit was directional, it remains 14 frames. If the hit was basic, then it remains 15 frames. Once the frame count is reached, the same decision logic applies as in S_BASIC_ATTACK_REC. The FSM will check the buttons and choose the states as if it were the idle state explained above.

A.4.7 Block-Stun State: The same logic for hit-stun applies also to the block-stun state. If the hit was directional, then it remains in a block-stun state for 12 frames. If the hit was basic, then it remains in a block-stun state for 13 frames.

A.4.8 Life/Block Shift Register Logic: We update the lives and blocks by using shift registers. When “life_control == HOLD”, the lives shift register holds its current 3-bit the same. When “life_control == SERIAL_L”, all bits shift to the left by one bit on the next clock edge. It drops the least significant bit to 0 when the player loses a life. The same logic applies to the remaining number of blocks by the left shift register. When the reset is high, each shift register is loaded with binary 111, which corresponds to 3 full lives or 3 full blocks. Moreover, all shift registers work with the same 60Hz clock (clk), which ensures that each life or block decrement is synchronized to the state change that triggers it.

B.Player Movement Module

In this module, since this is a 2D game, we only check for the player’s horizontal position (x-axis) on the screen. The player can move backward, forward, or maintain their position. These should be in parallel with the FSM states, and we need to convert these backward and forward commands into precise x-locations. However, players should not overlap with each other and stay inside the screen boundaries. All the location calculations are done by taking the origin as the top-left point of the screen. We will be explaining only Player 1’s movement since both logics are the same. The only difference is the coordinate difference. Simply, Player 2's movement is the mirror of Player 1's movement.

B.1 Inputs of the Player Movement Module: We have one 1-bit clock and one 1-bit reset signal, the same as in other modules. Two 4-bit Player1NS and Player2NS inputs are defined as state codes from each player's FSM. To have a position change, the character must be in either a forward state or a backward state. All other states in the FSM (Idle, attack, or stun) cause the player to remain stationary in x.

B.2 Outputs: Two 10-bit long output wires are called "Player1LocationsX0" and "Player2LocationsX0". These wires address the current locations of the players. Since we have a 640-pixel horizontal screen, 10 bits is enough to cover it.

B.3 Player 1 Movement: We split the movement of Player 1 into three forward, backward, and all other states.

B.3.1 Forward Movement: The module checks the following inequality to not overlap with player 2 in Eq.1:

$$P1_X + 3 * charW < P2_X - 1 - charW \quad Eq(1)$$

3*charW represents the minimum gap between the two players so that they do not overlap. By subtracting 1+charW, we will ensure that a 1-pixel separation is present between the two. When this inequality holds, Player 1 moves 3-pixel long. ("charW" corresponds to the character width, which is 32 pixels).

B.3.2 Backward Movement: When the player's location is less than the backward amount, which is 2 pixels, the player cannot go backward. Otherwise, it can go backward by a 2-pixel amount.

B.4 Reset Logic: When reset_n is high, both players start from their initial positions instantly. This reset logic works without considering their prior positions. When the reset is low, at each rising edge of the clock, the location of the players is stored in the Player1LocationsX and Player2LocationsX registers.

C. Hitscan Module

In this module, we decided to check if the player was hit or not by checking the player's current position and state.

C.1 Inputs & Outputs: We have four bits of input showing the player's current state. We have two nine-bit inputs showing the current position of the player. We have two hitscan outputs to determine the the player is hit or not.

C.2 Hitbox & Hurtbox Boundaries: We defined the player's boundaries. For player 1, we defined player 1's right edge by adding the player's width to the current position. For the second player, we defined the player 2's left edge by subtracting the player's width from the player 2's current position. With this definition, we assume that Player 1 attacks to the right, and Player 2 attacks to the left.

C.3 Hit Detection Logic: The first condition for Player 1 hitting Player 2 is that Player 1 is in the attack active state. The second condition is if Player 1's attack box overlaps Player 2's hurtbox. The second condition for Player 2 hitting Player 1 is that Player 2 is in the attack active state. The second condition is if Player 2's attack box overlaps Player 2's hurtbox. For each player, when either condition is satisfied, hitscan outputs 1. To summarize, we simply check the x-axis collisions and player states to determine the hitscan output.

D. Game1P & Game2P Modules

In this module, using the feedback of the FSM, Player Movement, and Hitscan modules, we organized the gameplay.

D.1 Inputs: We have three clock inputs 60Hz game logic clock, a 25MHz VGA clock, and a button clock. We have one active-low reset input. We have a 3-bit two-button input that comes from the users.

D.2 Outputs: We have 7-segment display outputs for showing the status of the gameplay (i.e., Game Over, Fight). We have pixel color output for VGA display to display the characters. We have a menu_can_start output to make the transition again to the menu when the game is over. We have lives1 and lives2 outputs to update the LEDs on the FPGA.

D.3 Area Logic: In this section, we have defined the character's hitbox and hurtbox areas. We defined the drawing regions player1_area, attack1_area, player2_area wires for selecting what color to draw, and assigned this info to the (next_x, next_y).

D.4 Text Drawing: In this section, we used the [1] pixel_on_text module for text drawing to the screen. We used to represent the lives, blocks of the players, a counter in the middle of the screen, and a game over status when the game was over.

D.5 Synchronising with Other Modules: In this section, we used the FSM, PlayerMovement, and Hitscan modules to decide the states of the character in the FSM, the player's location of the players based on the states.

D.6 Timer Display Logic: We have two counters for each binary digit. Each counts up to nine, starting from 0. By using the outputs of this logic and pixel drawer, we displayed the game counter on the VGA screen.

D.7 Game Over Logic: By using the outputs of the FSM and counter, we decided how the game ended. We used the "died" output of the FSM to decide which player wins. If player 1 dies, player 2 wins the game. If player 2 dies, player 1 wins the game. If both players die, the game ends in a draw. If the counter is 99, and none of the players have died, the game ends in a draw.

D.8 Game Restart Logic: We decide the restart logic by checking the user's confirmation for playing again. When the user presses a button, the game restarts to the game menu.

IV. RESULTS

Testbench results can be found in Appendix C.

Fig. 1. Game Menu Screen

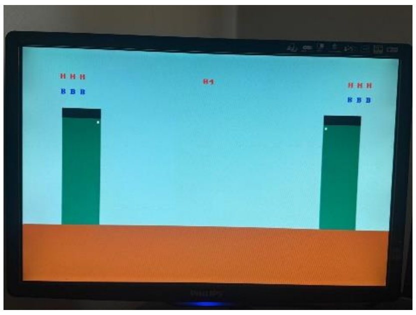


Fig.2 Idle State

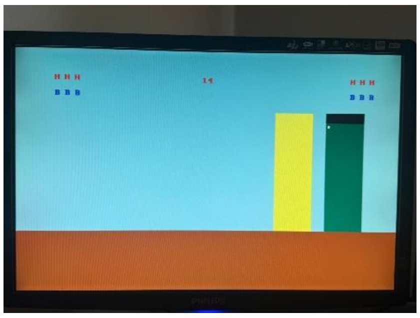


Fig.3 Basic Attack Start State

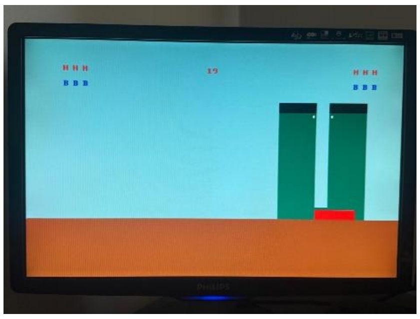


Fig.4 Basic Attack Active State

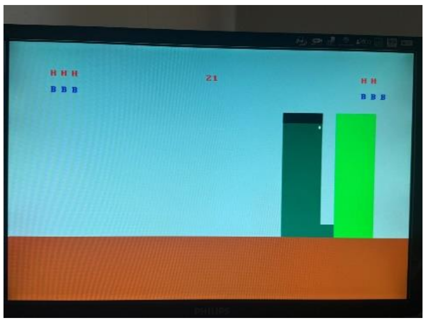


Fig.5 Player 1 Basic Attack Recovery & Player 2 Hitstun State

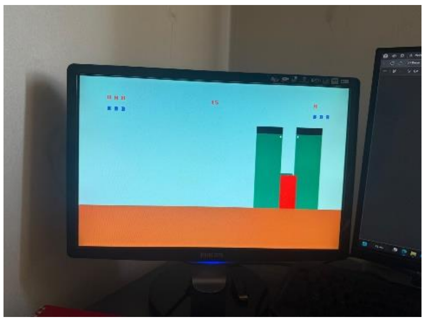


Fig.6 Basic Attack Active State

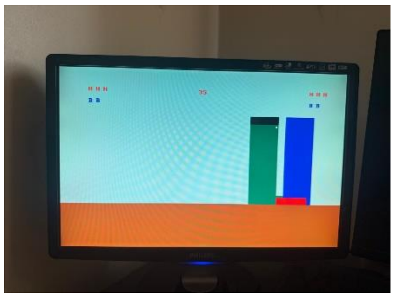


Fig.7 Block State

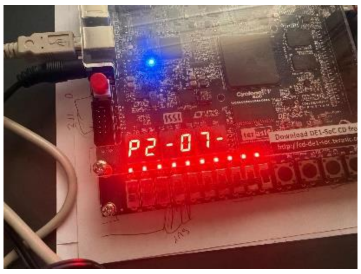
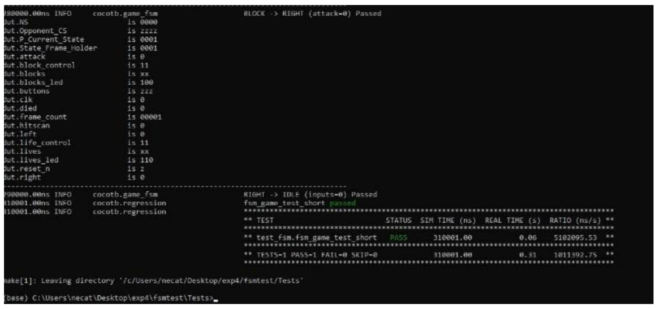


Fig.8 FPGA Winner & Timer Display


```
--------------------------------------------------------------------------------
280000.00ns INFO     cocotb.game_fsm                    BLOCK -> RIGHT (attack=0) Passed
dut.NS                        is 0000
dut.Opponent_CS               is zzzz
dut.P_Current_State           is 0001
dut.State_Frame_Holder        is 0001
dut.attack                    is 0
dut.block_control             is 11
dut.blocks                    is xx
dut.blocks_led                is 100
dut.buttons                   is zzz
dut.clk                       is 0
dut.died                      is 0
dut.frame_count               is 00001
dut.hitscan                   is 0
dut.left                      is 0
dut.life_control              is 11
dut.lives                     is xx
dut.lives_led                 is 110
dut.reset_n                   is z
dut.right                     is 0
--------------------------------------------------------------------------------
290000.00ns INFO     cocotb.game_fsm                    RIGHT -> IDLE (inputs=0) Passed
310001.00ns INFO     cocotb.regression                  fsm_game_test_short passed
310001.00ns INFO     cocotb.regression                  ******************************************************************************
                                                        ** TEST                           STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
                                                        ******************************************************************************
                                                        ** test_fsm.fsm_game_test_short   PASS       310001.00           0.06    5102095.53  **
                                                        ******************************************************************************
                                                        ** TESTS=1 PASS=1 FAIL=0 SKIP=0              310001.00           0.31    1011392.75  **
                                                        ******************************************************************************

make[1]: Leaving directory '/c/Users/necat/Desktop/exp4/fsmtest/Tests'

(base) C:\Users\necat\Desktop\exp4\fsmtest\Tests>
```
Fig.9 Testbench Result

V. Challenges and Discussion

While doing this project, we first do it module by module and solve the problems that we faced at the same time. However, after finishing all the modules, connecting and synchronizing them was really tough for us since every module had some time to wait or needed some flag for execution. Using multiple clock inputs and using them at the same time was also challenging for us. One physical challenge for this project is that solving the debouncing effect for the buttons that we bought online was also a hard thing to solve. Moreover, handling the edge cases like player collisions or some edge cases like when the two players hit each other at the same time, when they had both one remaining health, was very challenging and hard to decide what to do. Although we had lots of challenges during this project, solving them was quite made us feel the engineering satisfaction, and it improved our insight and perspective through the project.

VI. Conclusion

In conclusion, in this project, we successfully finished the project by using finite state machines (FSMs), pixel graphic rendering, and handling the inputs and outputs of the various modules. We had the chance to implement each component game menu, countdown, detailed gameplay logic, and visual output. By doing this project, we gained hands-on experience with Verilog HDL and deepened our digital design skills.

REFERENCES

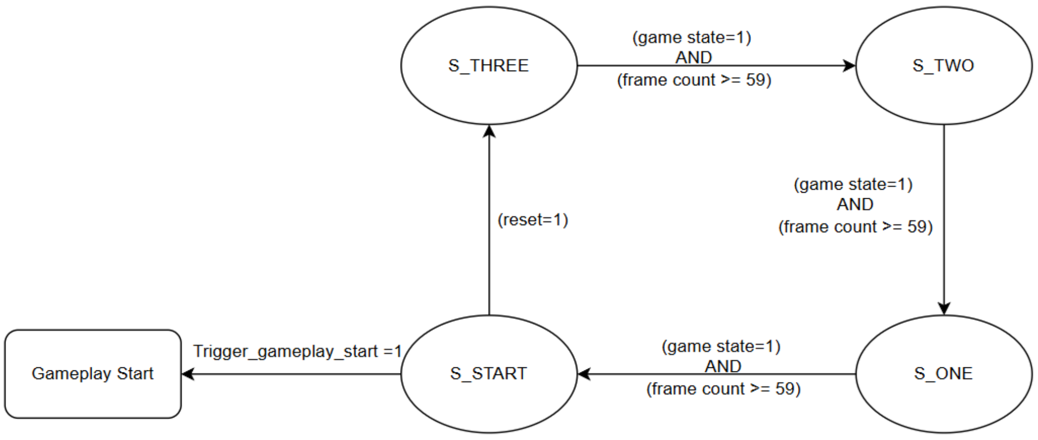
[1] Wang, D. X. (n.d.). *VGA Text Generator* [Source code]. GitHub. https://github.com/Derek-X-Wang/VGA-Text-Generator

[2] HiFight. (n.d.). *FOOTSIES: Rollback Edition*. https://hifight.github.io/footsies/

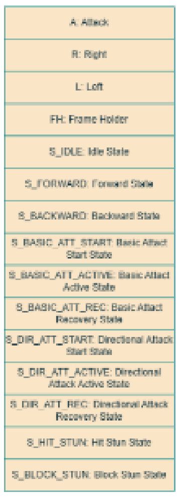
[3] Adams, V. H. (n.d.). *VGA driver for the DE1-SoC*. Retrieved May 30, 2025, from https://vanhunteradams.com/DE1/VGA_Driver/Driver.html

Appendices

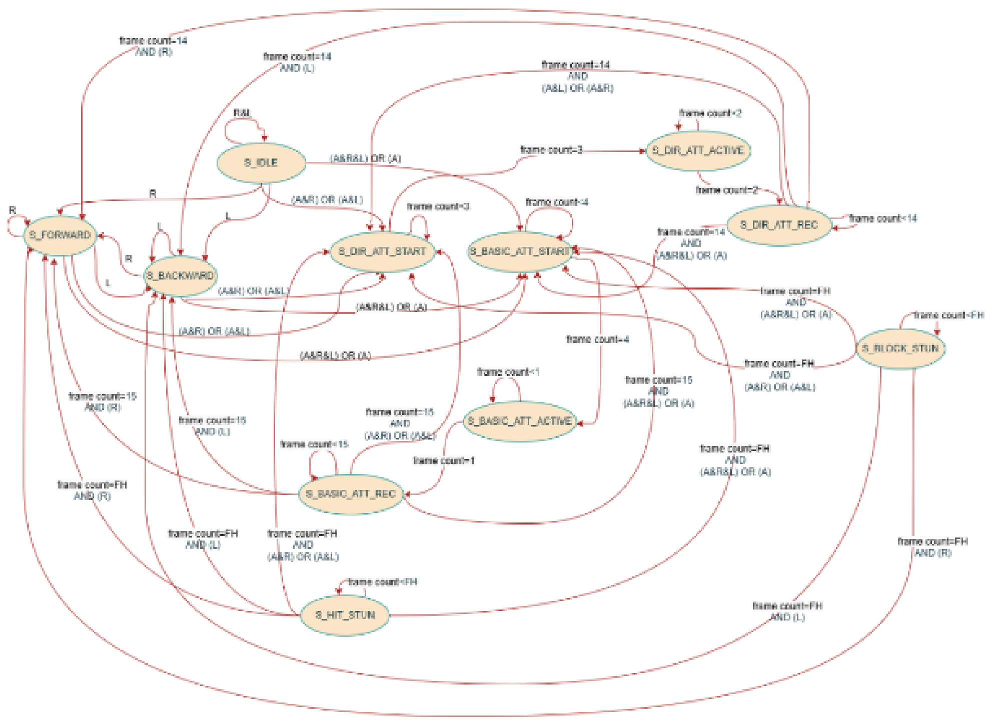
Appendix A


S_THREE
(game state=1) AND (frame count >= 59)
S_TWO
(reset=1)
(game state=1) AND (frame count >= 59)
Gameplay Start
Trigger_gameplay_start =1
S_START
(game state=1) AND (frame count >= 59)
S_ONE

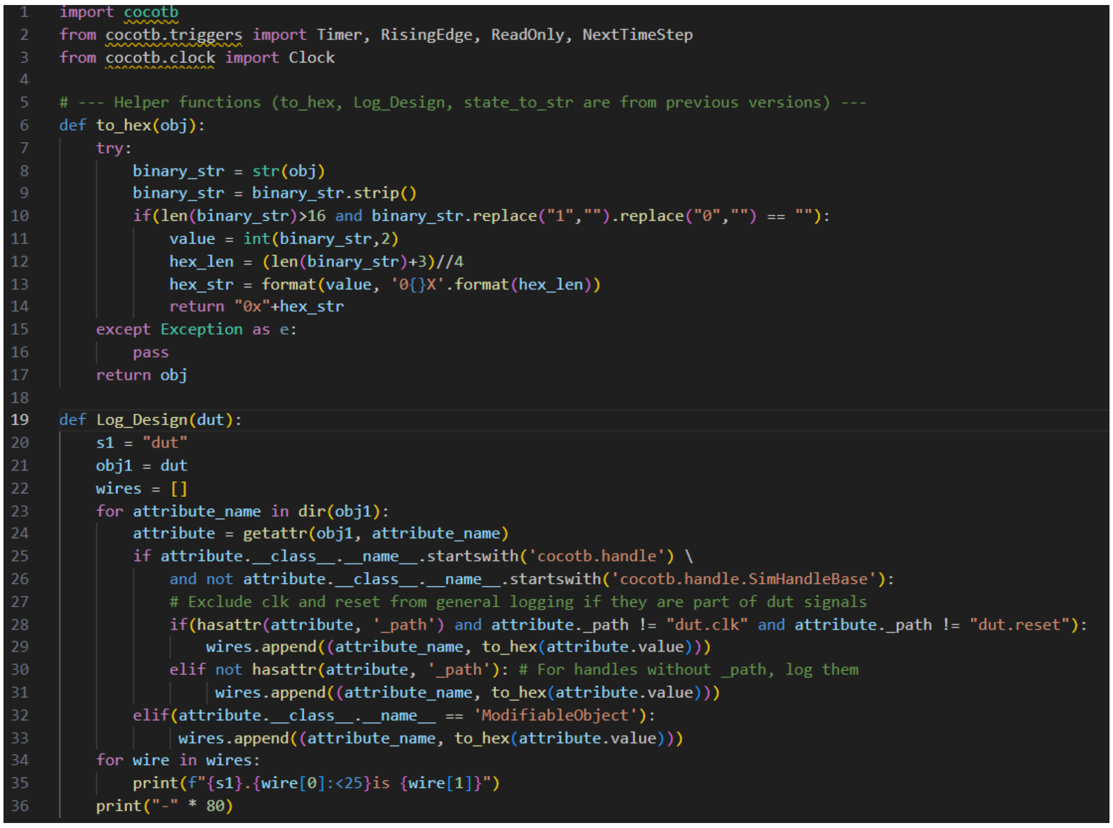
Appendix B



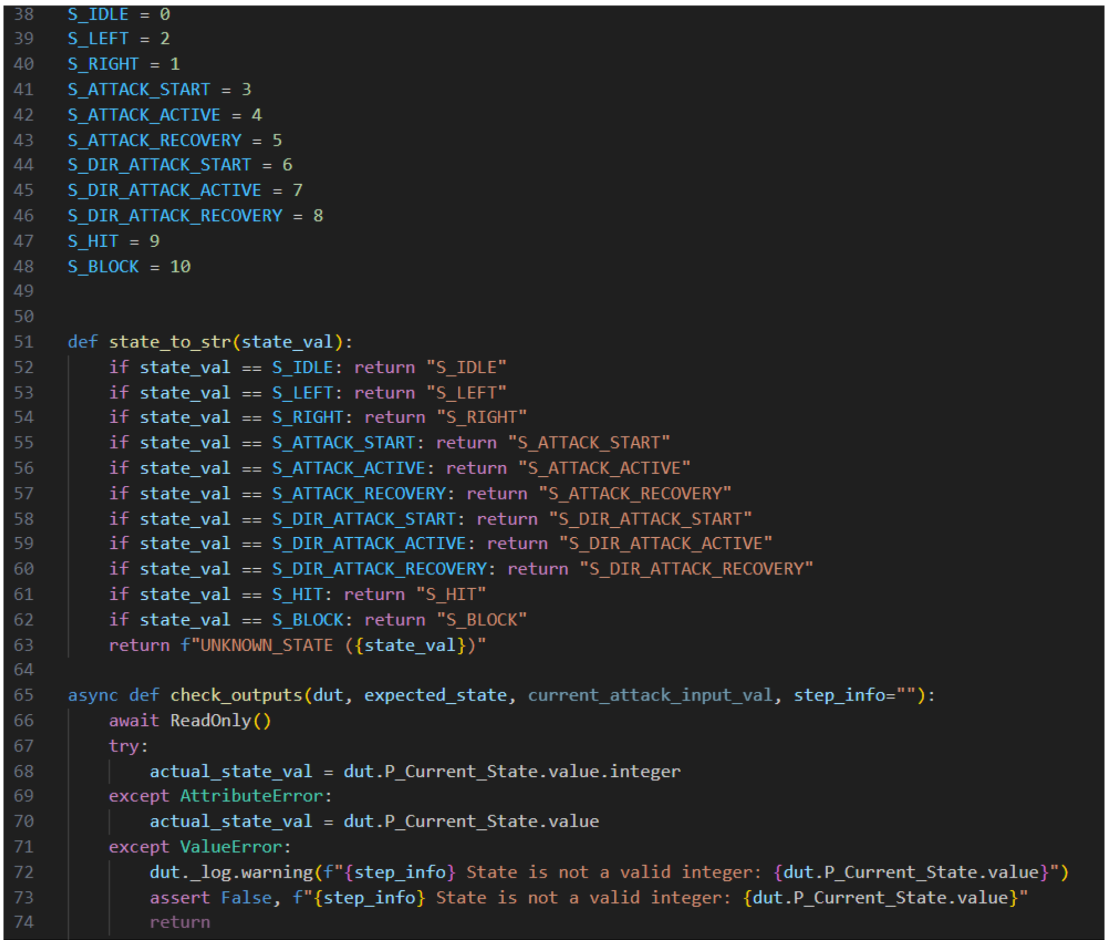
A: Attack
R: Right
L: Left
FH: Frame Holder
S_IDLE: Idle State
S_FORWARD: Forward State
S_BACKWARD: Backward State
S_BASIC_ATT_START: Basic Attack Start State
S_BASIC_ATT_ACTIVE: Basic Attack Active State
S_BASIC_ATT_REC: Basic Attack Recovery State
S_DIR_ATT_START: Directional Attack Start State
S_DIR_ATT_ACTIVE: Directional Attack Active State
S_DIR_ATT_REC: Directional Attack Recovery State
S_HIT_STUN: Hit Stun State
S_BLOCK_STUN: Block Stun State



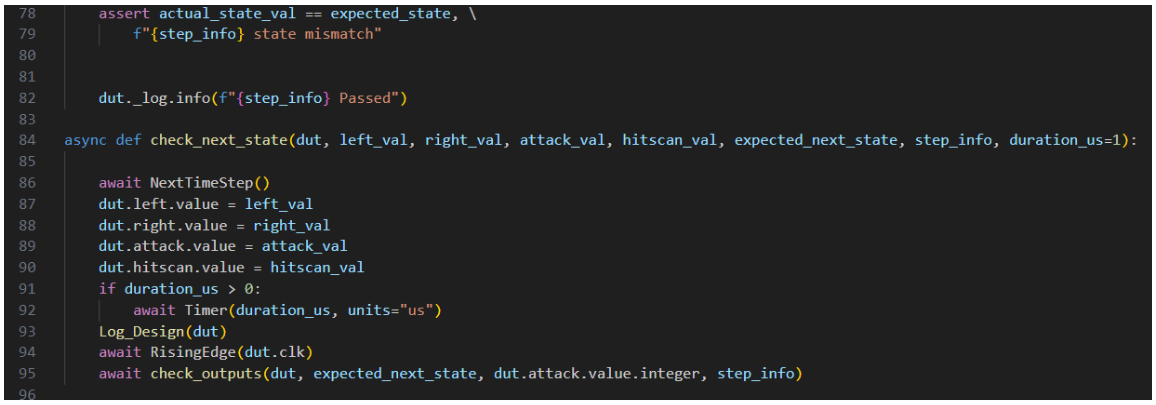
Appendix C



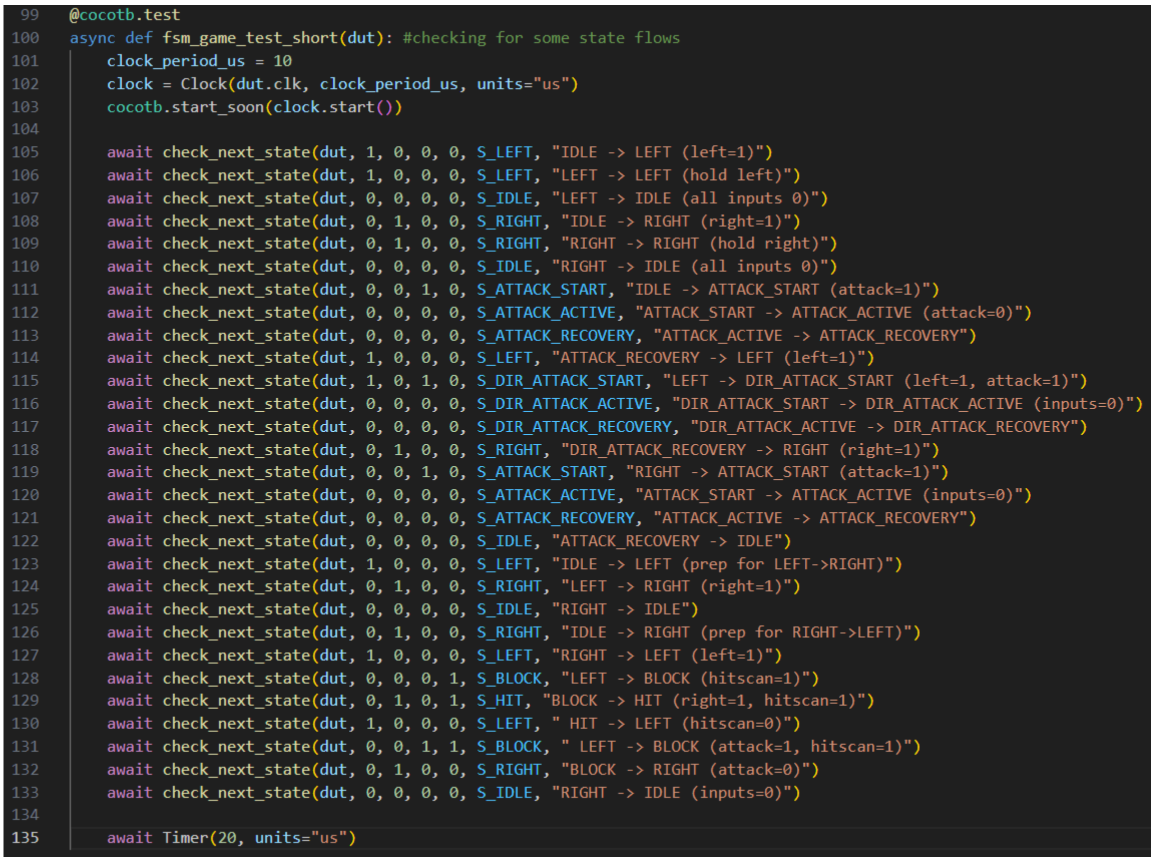
```python
import cocotb
from cocotb.triggers import Timer, RisingEdge, ReadOnly, NextTimeStep
from cocotb.clock import Clock

# --- Helper functions (to_hex, Log_Design, state_to_str are from previous versions) ---
def to_hex(obj):
    try:
        binary_str = str(obj)
        binary_str = binary_str.strip()
        if(len(binary_str)>16 and binary_str.replace("1","").replace("0","") == ""):
            value = int(binary_str,2)
            hex_len = (len(binary_str)+3)//4
            hex_str = format(value, '0{}X'.format(hex_len))
            return "0x"+hex_str
    except Exception as e:
        pass
    return obj

def Log_Design(dut):
    s1 = "dut"
    obj1 = dut
    wires = []
    for attribute_name in dir(obj1):
        attribute = getattr(obj1, attribute_name)
        if attribute.__class__.__name__.startswith('cocotb.handle') \
            and not attribute.__class__.__name__.startswith('cocotb.handle.SimHandleBase'):
            # Exclude clk and reset from general logging if they are part of dut signals
            if(hasattr(attribute, '_path') and attribute._path != "dut.clk" and attribute._path != "dut.reset"):
                wires.append((attribute_name, to_hex(attribute.value)))
            elif not hasattr(attribute, '_path'): # For handles without _path, log them
                wires.append((attribute_name, to_hex(attribute.value)))
        elif(attribute.__class__.__name__ == 'ModifiableObject'):
            wires.append((attribute_name, to_hex(attribute.value)))
    for wire in wires:
        print(f"{s1}.{wire[0]:<25}is {wire[1]}")
    print("-" * 80)
```



```python
S_IDLE = 0
S_LEFT = 2
S_RIGHT = 1
S_ATTACK_START = 3
S_ATTACK_ACTIVE = 4
S_ATTACK_RECOVERY = 5
S_DIR_ATTACK_START = 6
S_DIR_ATTACK_ACTIVE = 7
S_DIR_ATTACK_RECOVERY = 8
S_HIT = 9
S_BLOCK = 10


def state_to_str(state_val):
    if state_val == S_IDLE: return "S_IDLE"
    if state_val == S_LEFT: return "S_LEFT"
    if state_val == S_RIGHT: return "S_RIGHT"
    if state_val == S_ATTACK_START: return "S_ATTACK_START"
    if state_val == S_ATTACK_ACTIVE: return "S_ATTACK_ACTIVE"
    if state_val == S_ATTACK_RECOVERY: return "S_ATTACK_RECOVERY"
    if state_val == S_DIR_ATTACK_START: return "S_DIR_ATTACK_START"
    if state_val == S_DIR_ATTACK_ACTIVE: return "S_DIR_ATTACK_ACTIVE"
    if state_val == S_DIR_ATTACK_RECOVERY: return "S_DIR_ATTACK_RECOVERY"
    if state_val == S_HIT: return "S_HIT"
    if state_val == S_BLOCK: return "S_BLOCK"
    return f"UNKNOWN_STATE ({state_val})"

async def check_outputs(dut, expected_state, current_attack_input_val, step_info=""):
    await ReadOnly()
    try:
        actual_state_val = dut.P_Current_State.value.integer
    except AttributeError:
        actual_state_val = dut.P_Current_State.value
    except ValueError:
        dut._log.warning(f"{step_info} State is not a valid integer: {dut.P_Current_State.value}")
        assert False, f"{step_info} State is not a valid integer: {dut.P_Current_State.value}"
        return
```



```python
    assert actual_state_val == expected_state, \
        f"{step_info} state mismatch"


    dut._log.info(f"{step_info} Passed")

async def check_next_state(dut, left_val, right_val, attack_val, hitscan_val, expected_next_state, step_info, duration_us=1):

    await NextTimeStep()
    dut.left.value = left_val
    dut.right.value = right_val
    dut.attack.value = attack_val
    dut.hitscan.value = hitscan_val
    if duration_us > 0:
        await Timer(duration_us, units="us")
    Log_Design(dut)
    await RisingEdge(dut.clk)
    await check_outputs(dut, expected_next_state, dut.attack.value.integer, step_info)
```



```python
@cocotb.test
async def fsm_game_test_short(dut): #checking for some state flows
    clock_period_us = 10
    clock = Clock(dut.clk, clock_period_us, units="us")
    cocotb.start_soon(clock.start())

    await check_next_state(dut, 1, 0, 0, 0, S_LEFT, "IDLE -> LEFT (left=1)")
    await check_next_state(dut, 1, 0, 0, 0, S_LEFT, "LEFT -> LEFT (hold left)")
    await check_next_state(dut, 0, 0, 0, 0, S_IDLE, "LEFT -> IDLE (all inputs 0)")
    await check_next_state(dut, 0, 1, 0, 0, S_RIGHT, "IDLE -> RIGHT (right=1)")
    await check_next_state(dut, 0, 1, 0, 0, S_RIGHT, "RIGHT -> RIGHT (hold right)")
    await check_next_state(dut, 0, 0, 0, 0, S_IDLE, "RIGHT -> IDLE (all inputs 0)")
    await check_next_state(dut, 0, 0, 1, 0, S_ATTACK_START, "IDLE -> ATTACK_START (attack=1)")
    await check_next_state(dut, 0, 0, 0, 0, S_ATTACK_ACTIVE, "ATTACK_START -> ATTACK_ACTIVE (attack=0)")
    await check_next_state(dut, 0, 0, 0, 0, S_ATTACK_RECOVERY, "ATTACK_ACTIVE -> ATTACK_RECOVERY")
    await check_next_state(dut, 1, 0, 0, 0, S_LEFT, "ATTACK_RECOVERY -> LEFT (left=1)")
    await check_next_state(dut, 1, 0, 1, 0, S_DIR_ATTACK_START, "LEFT -> DIR_ATTACK_START (left=1, attack=1)")
    await check_next_state(dut, 0, 0, 0, 0, S_DIR_ATTACK_ACTIVE, "DIR_ATTACK_START -> DIR_ATTACK_ACTIVE (inputs=0)")
    await check_next_state(dut, 0, 0, 0, 0, S_DIR_ATTACK_RECOVERY, "DIR_ATTACK_ACTIVE -> DIR_ATTACK_RECOVERY")
    await check_next_state(dut, 0, 1, 0, 0, S_RIGHT, "DIR_ATTACK_RECOVERY -> RIGHT (right=1)")
    await check_next_state(dut, 0, 0, 1, 0, S_ATTACK_START, "RIGHT -> ATTACK_START (attack=1)")
    await check_next_state(dut, 0, 0, 0, 0, S_ATTACK_ACTIVE, "ATTACK_START -> ATTACK_ACTIVE (inputs=0)")
    await check_next_state(dut, 0, 0, 0, 0, S_ATTACK_RECOVERY, "ATTACK_ACTIVE -> ATTACK_RECOVERY")
    await check_next_state(dut, 0, 0, 0, 0, S_IDLE, "ATTACK_RECOVERY -> IDLE")
    await check_next_state(dut, 1, 0, 0, 0, S_LEFT, "IDLE -> LEFT (prep for LEFT->RIGHT)")
    await check_next_state(dut, 0, 1, 0, 0, S_RIGHT, "LEFT -> RIGHT (right=1)")
    await check_next_state(dut, 0, 0, 0, 0, S_IDLE, "RIGHT -> IDLE")
    await check_next_state(dut, 0, 1, 0, 0, S_RIGHT, "IDLE -> RIGHT (prep for RIGHT->LEFT)")
    await check_next_state(dut, 1, 0, 0, 0, S_LEFT, "RIGHT -> LEFT (left=1)")
    await check_next_state(dut, 0, 0, 0, 1, S_BLOCK, "LEFT -> BLOCK (hitscan=1)")
    await check_next_state(dut, 0, 1, 0, 1, S_HIT, "BLOCK -> HIT (right=1, hitscan=1)")
    await check_next_state(dut, 1, 0, 0, 0, S_LEFT, " HIT -> LEFT (hitscan=0)")
    await check_next_state(dut, 0, 0, 1, 1, S_BLOCK, " LEFT -> BLOCK (attack=1, hitscan=1)")
    await check_next_state(dut, 0, 1, 0, 0, S_RIGHT, "BLOCK -> RIGHT (attack=0)")
    await check_next_state(dut, 0, 0, 0, 0, S_IDLE, "RIGHT -> IDLE (inputs=0)")

    await Timer(20, units="us")
```