數位電路實驗
Final Project
Greedy Snake

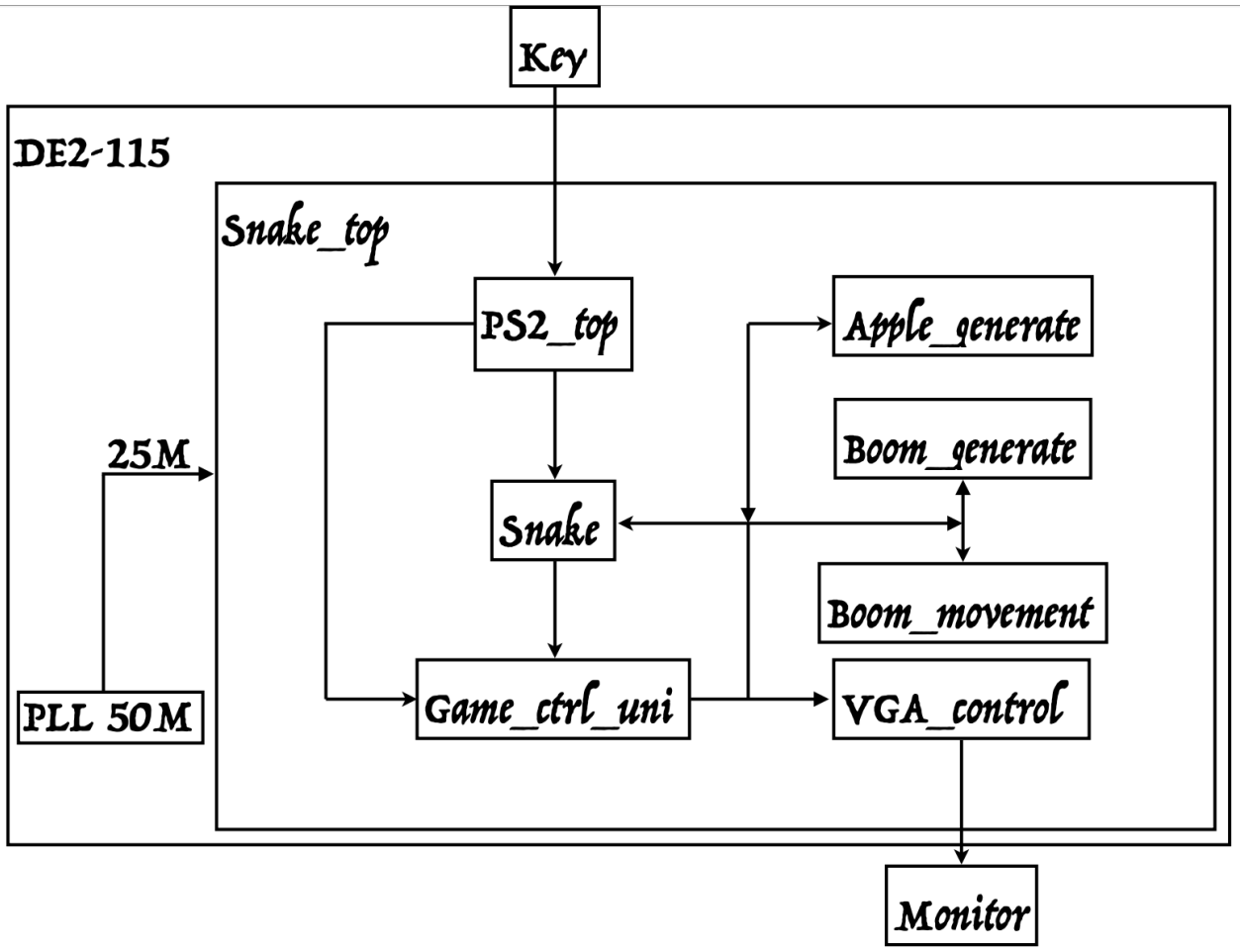
Team 02

金家逸 B10502076
王維勤 B10502010
林桓鈺 B10502013

1. Introduction:

"Greedy Snake" is a classic nostalgic game that originated in 1976. The game objective is straightforward: players need to control a snake to eat randomly generated food. Each time the snake consumes food, its body grows longer. The challenge of the game lies in keeping the snake alive and making it as long as possible. In our modified version of the game, we have transformed it into a two-player version. In addition to implementing its basic functionalities, we have also added some of our own creative elements, such as the "multiple apples" feature and the "bomb" function, enhancing the gameplay of the two-player mode.

2. System Architecture:



(1) **DE2_115.sv:** The module used to connect with the DE2-115 FPGA board.

(2) **Debounce.sv:** Used for debouncing keys 0 to 3.

(3) **snake_top.sv:** The top module used to connect signals with the DE2-115 and various submodules.

(4) **snake.sv:** Implementation of various logic related to the snake.

1. Speed Control:
   - ➢ Including control over the snake's movement speed. The speed increases linearly based on the total number of apples the snake has consumed.
2. Collision Handling:
   - ➢ If the snake collides with its own body or the body of another snake, it dies.
   - ➢ If the body's front three segments collide with a bomb, the snake dies.
3. Direction of Movement:
   - ➢ For each movement, the position of the previous segment is assigned to the current segment.
4. Length Adjustment:
   - ➢ The snake's body length increases when it consumes apples.
   - ➢ The snake's body is truncated when it collides with a bomb.
5. Rendering:
   - ➢ Rendering the snake's image at the corresponding position on the screen.

(5) **game_ctrl_unit.sv:** Used to control various states of the game from start to finish. The game has four states:

1. REPLAY:
   - ➢ When the restart signal is input, the screen displays the welcome interface.
   - ➢ After 2 seconds, it transitions to the START state.

2. START:

- ➢ Displays the snake and the map on the screen.
- ➢ When either player presses the control key, the game begins, transitioning to the PLAY state.

3. PLAY:

- ➢ Both players start the game.
- ➢ When a player dies, there is a 2-second pause before entering the DIE state.

4. DIE:

- ➢ Displays which player has won.
- ➢ Pressing any directional key returns to the RESTART state.

(6) VGA_control .sv: Divide the screen into a grid of 40x30 small squares, with each square being 256 pixels in size. This grid is utilized to render various images and text onto the screen, including apples, bombs, the heads, bodies, and tails of the two snakes, among other elements. Text is employed to display post-game prompts indicating which player emerged victorious.

(7) PS2.sv: A module used to control a PS2 keyboard, utilizing the keys on the keyboard, specifically (W, A, S, D) and (I, J, K, L). Receiving signals from a PS2 keyboard to control the movement of a snake in the directions of up, down, left, and right. The module outputs two sets of signals, one representing the keys pressed by player one (W, S, A, D), and the other representing the keys pressed by player two (I, K, J, L). The module incorporates two sets of registers to store the inputs for each player, adjusting their content based on the start and end signals from the PS2 keyboard, effectively indicating the keys pressed by the players.

(8) apple_generate.sv: A module designed to generate three apples and determine whether the two snakes have consumed the apples. If the coordinates of a snake's head match the coordinates of an apple, it is considered that the snake has consumed the apple. After a few seconds, another apple is randomly generated. The randomness in this context is

achieved by using an initial value and continuously adding a specific number, creating a pseudo-random number effect.

(9) **boom_generate.sv:**When "ready_next_boom" is received, a new bomb position is generated using random numbers. By utilizing information about the snake's head position, we can determine which snake has come into contact with the bomb. After the bomb is triggered, "o_boom_display" is set to 1. If player one comes into contact with the bomb, "o_boom_active1" is set to 1; if player two comes into contact with the bomb, "o_boom_active2" is set to 1.

(10) **boom_movement.sv:**After the game starts, it enters the "S_WAIT" state. When "boom_active1" assigns the position of player one's snake head to three bombs, to prevent the snake head from directly contacting the bomb and causing instant death, the bombs are pre-moved by one grid. If "boom_active2" is one, the process is the same, assigning the initial position of the bombs and then entering the "S_ACT" state.

In the "S_ACT" state, the movement direction of each bomb is determined based on the snake's forward direction. If the snake is moving upward, bomb 1 moves in the negative x-axis direction, bomb 2 moves in the negative y-axis direction, and bomb 3 moves in the positive x-axis direction. Next, check if each bomb is out of bounds. When the position of bomb 1 is out of bounds and "boom_display" is 0, set "outside1" to 1. When all "outside" values are 1, set "o_ready_next_boom" to 1, notifying "boom_generate.sv" to generate the next bomb, and then return to the "S_WAIT" state..

(11) **apple_image_rom.sv:** A module used to calculate the coordinates for storing the apple image in ROM. Providing a VGA control for displaying apple image. The VGA control receives the current scanned pixel point, while the ROM, based on the given x, y coordinates, performs the necessary conversion to obtain the address required for reading the image. As the screen

is divided into 16x16 grids, each displaying a picture, the address conversion can be achieved by extracting the last four bits.

(12) snake_image_rom.sv: Calculate address for image of snake head, body, and tail. By utilizing reverse coordinate reading, achieve left-right and up-down mirroring effects, allowing the four different directions of the body to be represented without the need to store four separate images, thereby saving hardware resources.

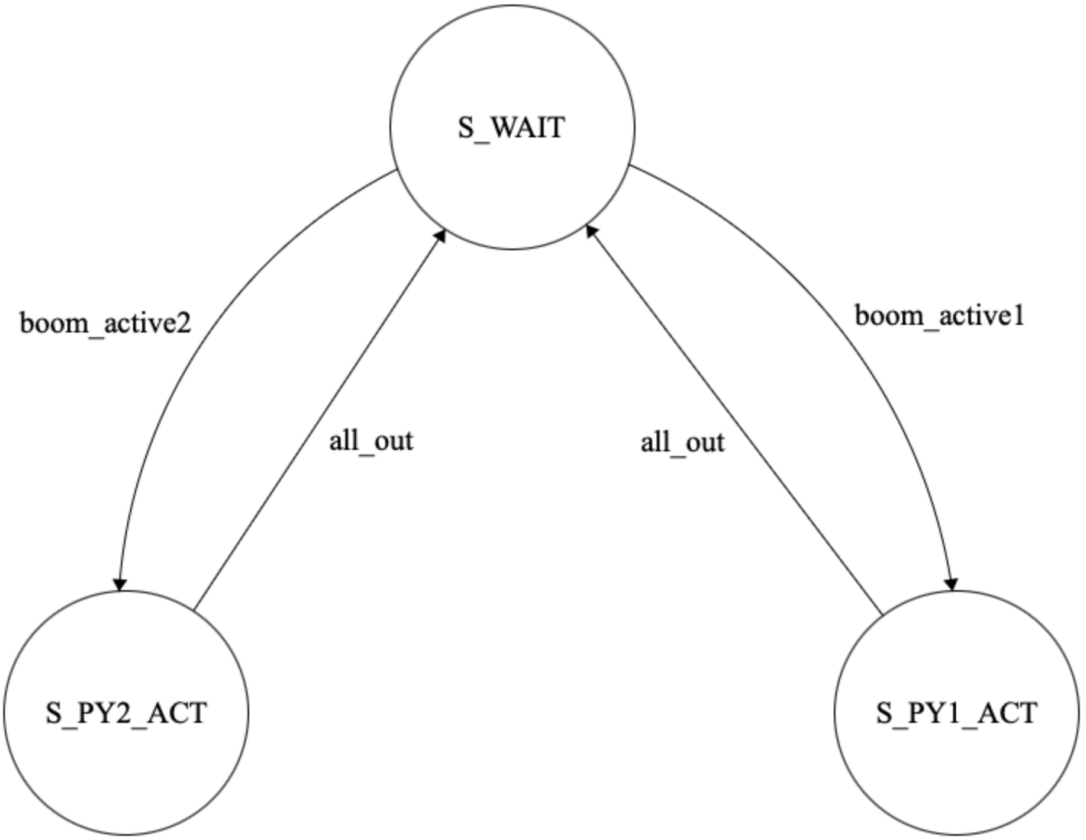
(13) resources.bomb_image_rom.sv: Calculate address for bomb image.

(14) *_data.sv: Providing the image file. It is implemented using a look-up table approach. Using C++, the image's MIF (Memory Initialization File) is transformed into corresponding Verilog code.

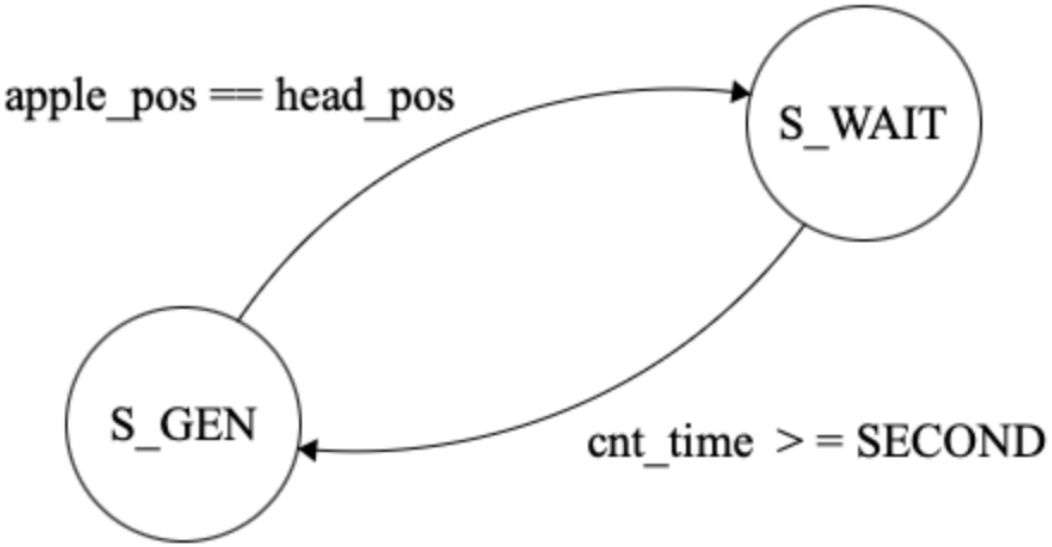
(15) score_ctrl.sv: A module designed to calculate the total number of apples consumed by two snakes. The calculated value is then used to control the speed of snake movement, with the speed increasing as the snakes consume more apples.

3. Finite state machine:

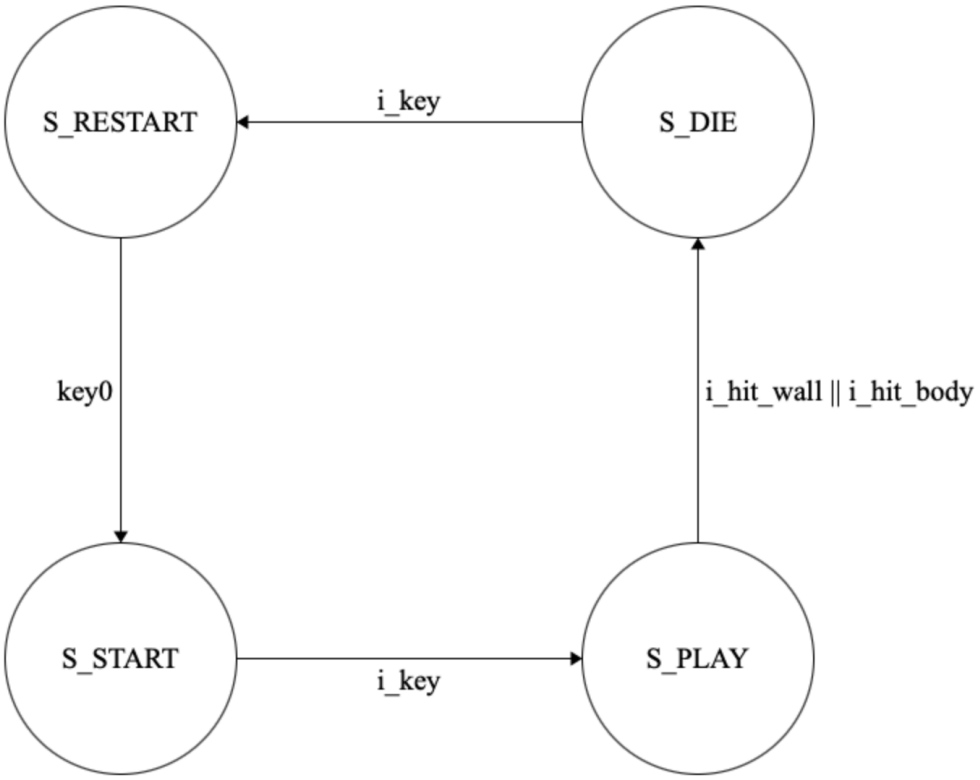
a. boom_movement.sv:



b. apple_generate.sv :



c. game_ctrl_unit.sv



4. File Structure:

```
team02_final/
├── team02_final_report
└── src/
      ├── apple.v
      ├── apple.qip
      ├── boom.v
      ├── boom.qip
      ├── snake_qsys.v
      ├── snake_qsys.qip
      ├── DE2_115.sv
      ├── apple_generate.sv
      ├── boom_generate.sv
      ├── boom_movement.sv
      ├── PS2.sv
      ├── score_ctrl.sv
      ├── snake.sv
      ├── VGA_control.sv
      ├── game_ctrl_unit.sv
      ├── Debounce.sv
      ├── apple_image_rom.sv
      ├── bomb_image_rom.sv
```

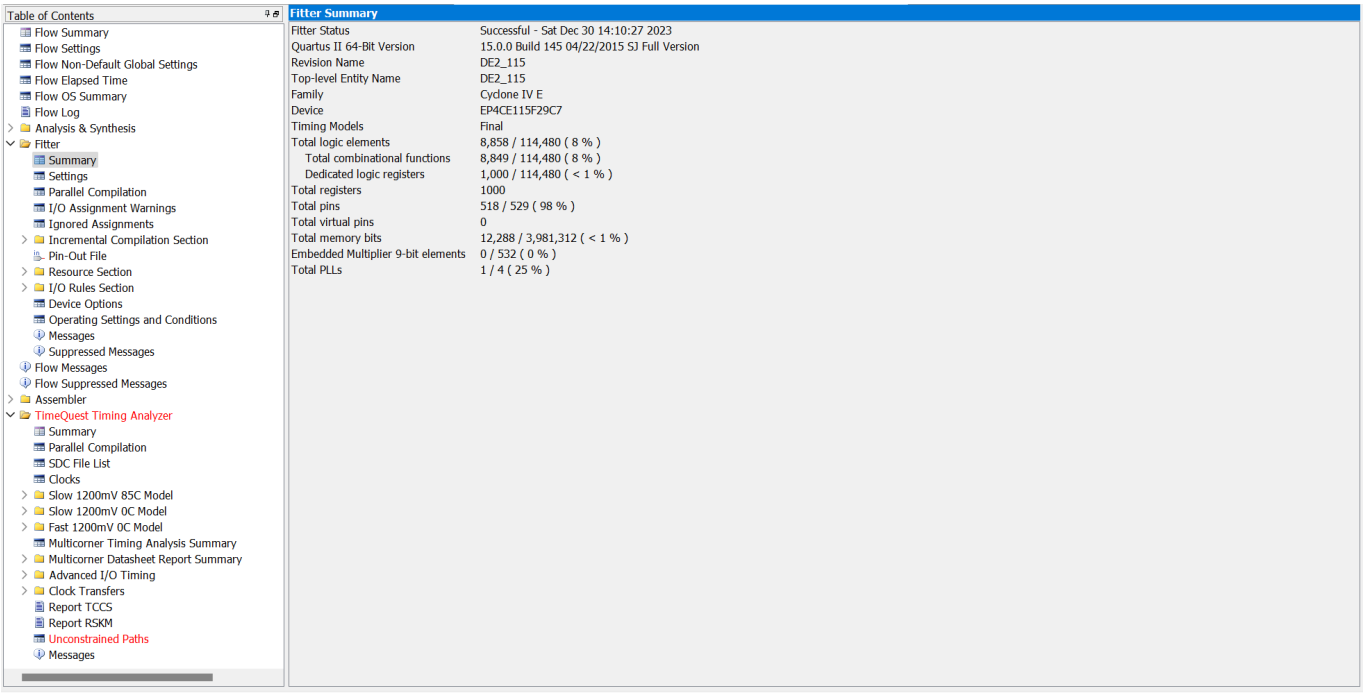
```
├── snake_image_rom.sv
├── green_body_data.sv
├── green_head_data.sv
├── green_tail_data.sv
├── green_UL_data.sv
├── green_UR_data.sv
├── red_body_data.sv
├── red_head_data.sv
├── red_tail_data.sv
├── red_UL_data.sv
├── red_UR_data.sv
└── snake_top.sv
```

5. usage steps:

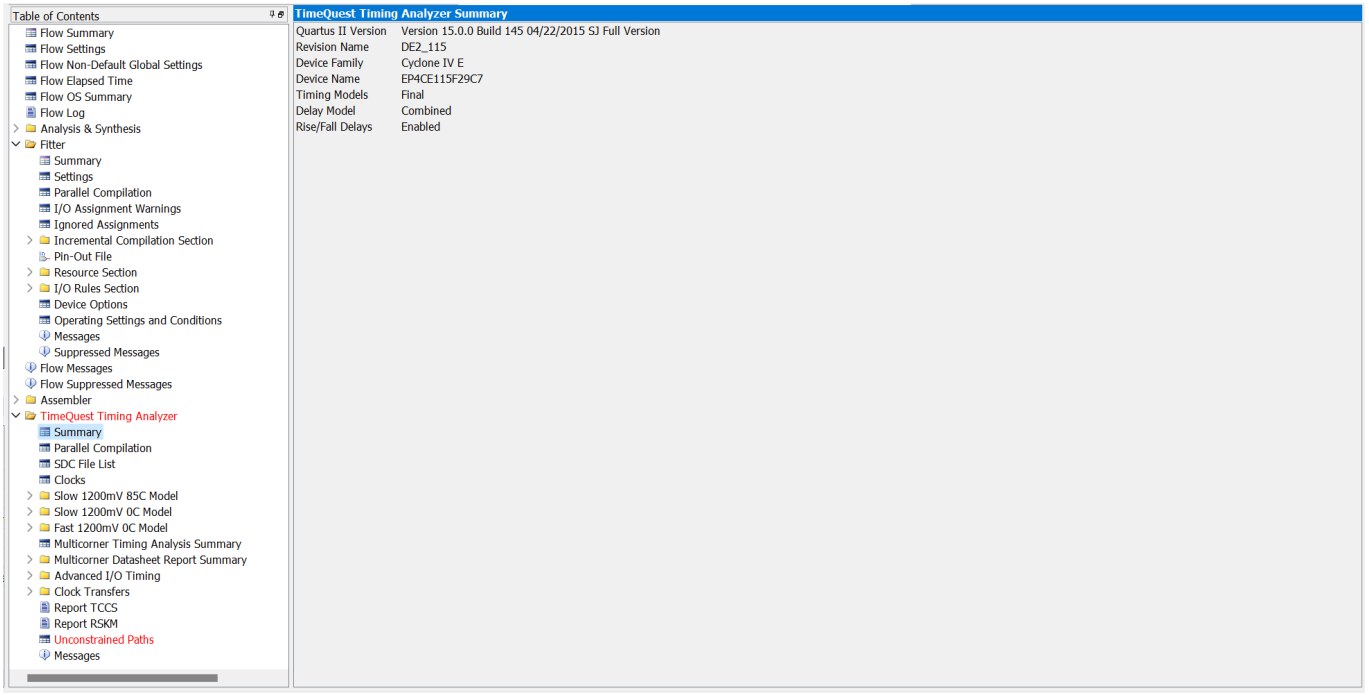
(1) 把SW[3]、SW[4]拉成high，並且按下key0進入遊戲畫面。
(2) 只要其中一個玩家按下任意鍵(W,A,S,D,I,J,K,L)遊戲就開始。
(3) 在遊戲期間，一旦其中玩家撞到牆壁或是撞到自己身體，或是撞到對手的身體，或是被炸彈炸死，遊戲及結束。
(4) 遊戲結束後，螢幕會顯示哪個玩家獲勝，此時按(W,A,S,D,I,J,K,L)任意鍵，會重新回到遊戲畫面。
(5) 重複上述流程直到把SW[3]拉成low。

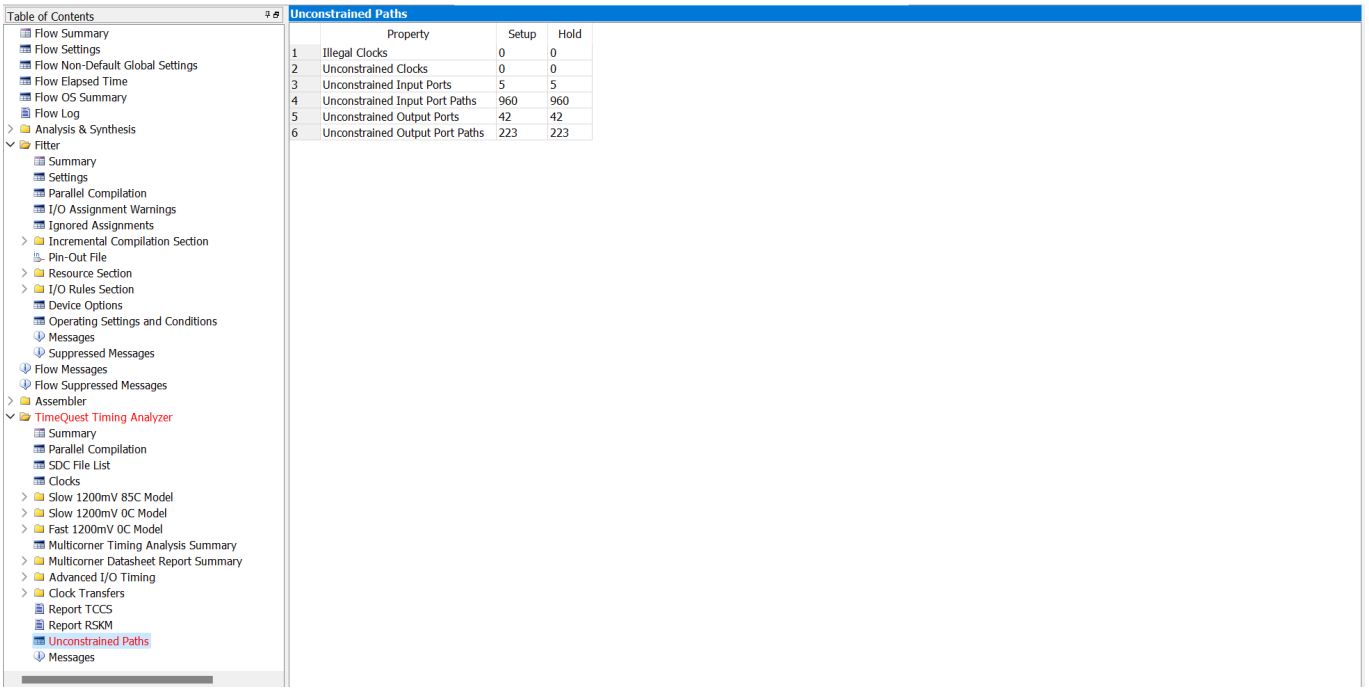
6. screenshot:

(1) Fitter Summary



(2) Timing Analyzer





7. Problems Encountered and Solutions:

a. At the beginning, our team intended to create a radio for our project. However, we encountered difficulties in successfully filtering out the signals from the external antenna. Due to this challenge, we considered using a pre-built radio module and decided to write our

own I2C code to initialize that module. Despite our persistent efforts, we were unable to make it work. After much frustration and discussion, our team decided to change the project one week before the final deadline and opted to create the game described.

b. During the implementation of the game, the most time-consuming aspect for us was figuring out how to successfully display images on the screen through VGA. Initially, we used Quartus' built-in ROM IP core to store our .mif images. However, the rendered images were consistently cut off, causing misalignment. We speculated that the issue might be due to delays in reading data from the ROM, leading to coordinates on the VGA display not matching the original rendering coordinates. As a solution, we manually implemented a ROM to store various image information. This eliminated delay issues when reading data through addresses.
c. This final project has deepened our awareness of the areas in FPGA that our team is unfamiliar with. FPGA contains various modules and IOs that require in-depth research to make them function successfully. Although the result we achieved in the end was not the initial radio project we planned, we learned valuable lessons in efficient teamwork and task allocation during the process. Additionally, we acquired knowledge of PS2 and VGA-related protocols, which were new to us and not covered in previous labs. While we may not be fully proficient in FPGA, we have gained a foundational understanding. In the future, we plan to delve further into its internal principles to address the shortcomings of this final project.