Final Project. Snack Game

2023. 12. 15.

21800689 조성민

22100355 서민재

목차

[1] 문제 정의

1. 프로젝트 목표

프로젝트의 목표는 Nexys-A7 보드를 사용하여 스네이크 게임을 개 발하는 것입니다. 이 게임에서 플레이어는 키보드를 사용하여 화면상의 스네이크를 제어합니다. 스네이크가 무언가를 먹을 때마다 길이가 길어지고, 스네이크가 자신의 몸이나 벽에 부딪히지 않도록 하는 것이 중요합니다. 게임은 플레이어의 반응 시간 과 전략적 사고를 테스트합니다.

2. 프로젝트의 기능

(1) 스네이크 게임 메커니즘 설계:
게임의 핵심은 플레이어가 키보드를 사용하여 스네이크를 조종하고, 먹이를 먹으면서 스네이크의 길이를 늘리는 것이다. 이때 자기 자신에게 부딪히거나 벽에 부딪힐 경우, 스네이크는 죽고, 게임은 끝이 난다. 게임은 3가지 모드를 설정할 수 있습니다.

보드를 실행 시, 게임모드는 실행 준비 상태입니다. – 5초간 화면이 뜹니다.

5초 후 대기모드로 들어갑니다. 이때 대기모드는 키보드 입력이 있을 때까지 유지됩니다.

키보드 입력을 받아들인 후, 게임이 시작됩니다. (단, 키보드 입력은 방향키만 사용가능)

방향키를 이용해서 자유롭게 스네이크를 조절 가능합니다. (단, 현재 진행방향과 반대방향으로는 이동불가, 이동 가능 시 바로 죽기 때문.)

게임을 플레이하면서 사과를 먹습니다. 사과를 먹는경우 스네이크의 꼬리는 길어집니다. (단 32개의 꼬리까지 길어지며 그 이후로는 길어지지 않습니다)

주의사항: 게임 모드를 설정하지 않으면 실행이 되지 않게 설정되어 있다.

(2) VGA 출력을 통한 그래픽 인터페이스 구현: Nexys-A7 보드의 VGA 출력 기능을 활용하여 게임의 시각적 요소를 구현합니다. 이에는 스네이크, 배경, 등이 포함됩니다.

(3) 사용자 입력 처리: 키보드를 통한 사용자 입력을 실시간 처리하고, 게임 로직에 통합한다.

(4) 오디오 통합: 게임은 음악과 사운드 이펙트를 통합하여 몰입감 있는 경험을 제공한

다.

3. 사용된 i/O

A. VGA 출력: 게임의 시각적 요소는 VGA 인터페이스를 통해 모니터에 출력됩 니다.

B. 키보드 입력: 플레이어는 키보드를 사용하여 게임 내에서 스네이크의 움직 임을 제어합니다.

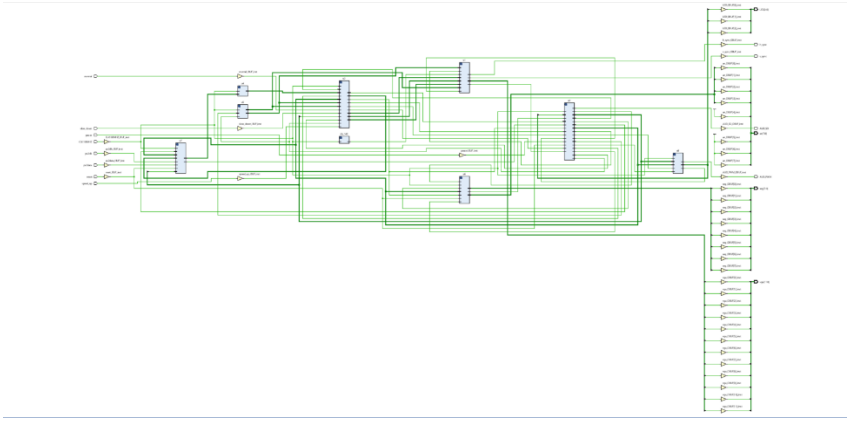
C. 오디오 출력: 게임은 배경 음악과 사운드 이펙트를 제공합니다. D. 스위치: Nexys-A7 보드에 내장된 스위치는 게임 설정 조정 (speed 컨트롤) 이나 특정 게임 컨트롤에 사용될 수 있습니다.

E. 7seg 출력: 게임의 중요 요소인 점수 관련을 7seg를 통해 출력됩니다.

F. Button 입력: 플레이어는 BTNC를 통해 게임을 reset 하고 원래 자리로 돌릴 수 있다.

[2] 설계

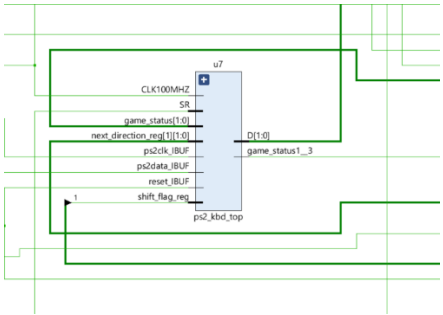
해당 FPGA 설계에서는 **Ps2_kbd_top, turn, apple**, **snake**, **display, score**, **fsm, audio** 등의 주요 모듈들이 각각의 역할을 수행하며, 뱀 게임의 다양한 기능을 담당합니다. 예를 들어, **Ps2_kbd_top** 모듈은 키보드 입력을 처리하고, **snake** 모듈에 방향 제어 신호를 전달하며, **snake** 모듈은 **display** 모듈과 상호작용하여 게임의 시각적 출력을 화면에 업데이트합니다. **fsm** 모듈은 게임 상태를 제어하며, 게임이 시작, 일시 정지, 종료 등의 상태를 조건에 따라 관리합니다. **Audio** 모듈은 게임 상태에 대하여 조건에 따라 각각 다른 소리를 출력합니다.



<전체 블록도>

1. Ps2_kbd_top

Ps2_kbd_top 모듈은 PS/2 키보드에서 입력 신호(**ps2clk, ps2data**)를 받아들여 처리합니다. 모듈의 출력 신호 중 D는 **turn**모듈과 game_status는 **fsm** 모듈에 연결됩니다. **turn**모듈은 사용자의 키보드 입력에 따라 게임 내 뱀의 방향을 조정하는 데 사용되며, **fsm**모듈은 게임의 전반적인 상태를 제어하는 데 사용됩니다.

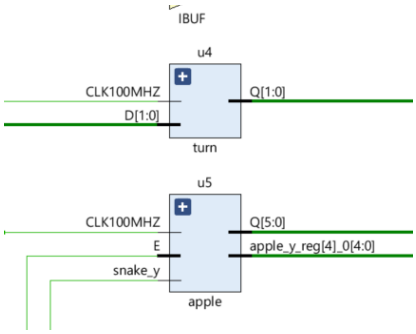


<ps2_kbd_top 모듈>

2. turn, apple

turn 모듈은 사용자의 키보드 입력에 따라 뱀의 방향을 결정합니다. 이 모듈은 **Ps2_kbd_top** 모듈로부터 방향 제어 신호를 받고, 이를 **snake**모듈에 전달하여 게임 내 뱀의 움직임을 제어합니다.

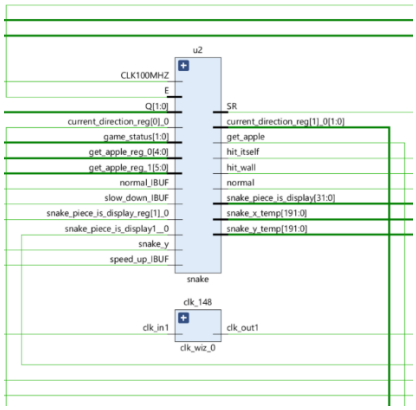
apple모듈은 게임 내에서 사과의 위치를 무작위로 생성합니다. 이 모듈은 게임의 진행 상태를 기반으로 작동하며, 생성된 사과의 위치는 **display**모듈과 **snake**모듈에 전달되어 화면에 표시되고, 뱀의 움직임에 영향을 미칩니다.



<turn & apple 모듈>

3. snake

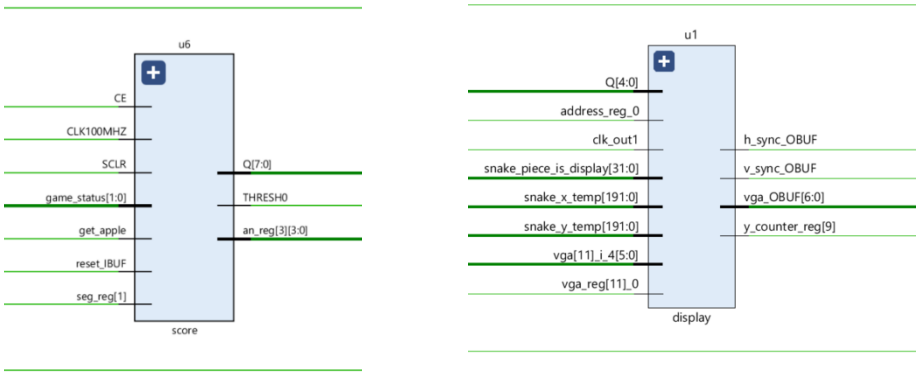
snake모듈은 게임 내 뱀의 움직임을 관리하는 중요한 부분으로, 여러 입력 신호를 받아들입니다. 이 모듈은 **turn**모듈로부터 뱀의 이동 방향을 결정하는 신호를 받으며, **apple**모듈에서 사과의 위치 정보를 제공받아 사과를 먹었는지 판단합니다. 또한, **fsm**모듈로부터 게임의 현재 상태에 대한 정보를 받아 게임의 진행을 조절합니다. 이 모듈의 출력은 **display**모듈에 전달되며, 뱀의 현재 위치와 상태 정보를 화면에 표시하는 데 사용됩니다. 이러한 입력과 출력 신호들은 게임의 데이터 경로와 제어부 간의 상호작용을 나타냅니다.



<snake 모듈>

4. display, score

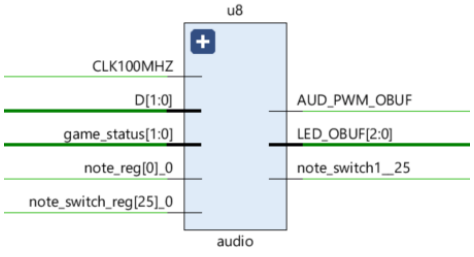
display모듈은 게임의 그래픽 출력을 담당합니다. 이 모듈은 **snake**모듈로부터 뱀의 위치 및 상태 정보, 그리고 **apple**모듈로부터 사과의 위치 정보를 입력으로 받습니다. 이 정보를 바탕으로 게임 화면을 구성하고 VGA 출력을 통해 화면에 표시합니다.
score모듈은 게임의 점수를 계산하고 표시합니다. 이 모듈은 **snake**모듈로부터 사과를 먹은 정보를 입력으로 받아 점수를 계산합니다. 계산된 점수는 7-segment 디스플레이를 통해 표시되며, 게임의 진행 상태에 따라 점수 표시를 업데이트합니다.



<score, display 모듈>

5. audio

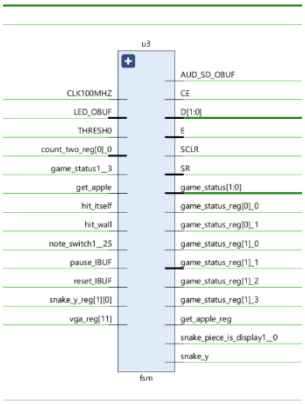
audio모듈은 100MHz 클럭 신호 (CLK100MHZ)와 게임의 상태 (game_status)를 입력으로 받아 게임의 오디오 출력을 관리합니다. 이 모듈은 AUD_PWM, AUD_SD, 그리고 LED와 같은 출력 신호를 제어합니다. 게임 상태에 따라 오디오 및 LED 출력을 조절하는 논리가 이 모듈 내부에 구현되어 있으며, 게임 상태 변경 시 오디오 및 LED 출력을 업데이트합니다.



<audio 모듈>

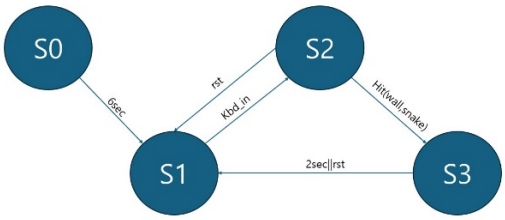
6. fsm

fsm모듈은 게임의 상태를 관리하고 제어합니다. 이 모듈은 네 가지 주요 상태 LAUNCHING, PLAYING, DIE_FLASHING, INITIALIZING을 갖고 있습니다. Reset 입력이 활성화되면 게임은 INITIALIZING상태로 초기화됩니다. LAUNCHING 상태에서는 일정 시간이 지나면 INITIALIZING상태로 전환되며, PLAYING 상태에서는 벽이나 자신에게 부딪히면 DIE_FLASHING 상태로 전환됩니다. DIE_FLASHING 상태에서도 일정 시간 후 INITIALIZING 상태로 돌아갑니다. 마지막으로, INITIALIZING 상태에서 방향키 입력이 들어오면 PLAYING 상태로 변경되어 게임이 시작됩니다



<fsm 모듈>

Fsm 모듈을 다이어그램으로 표현하면 다음 이미지로 표현하면 다음과 같이 표현할 수 있습니다.



<fsm 다이어그램>

[3] 시뮬레이션 결과

```
// Test stimulus
initial begin
    // Initialize inputs
    reset = 1;
    pause = 0;
    slow_down = 0;
    speed_up = 0;
    normal = 0;
    ps2clk = 1; // PS/2 clock high by default
    ps2data = 1; // PS/2 data high by default

    // Reset the system
    #100;
    reset = 0;
    #100;
    reset = 1;

    // Simulate pressing the down arrow key
    // Send Extended key prefix (E0)
    send_ps2_key(8'hE0);
    // Send down arrow key code (72)
    send_ps2_key(8'h72);

    // Additional delay after sending the keys
    #1000;
end
```

```
// Task to send a key code according to the PS/2 protocol
task send_ps2_key;
    input [7:0] keycode;
    integer i;
    reg odd_parity;
begin
    // Start bit (0)
    ps2data = 0;
    ps2clk = 0;
    #100;
    ps2clk = 1;
    #100;

    // Sending 8-bit data (LSB first)
    odd_parity = 1; // Initialize for odd parity calculation
    for (i = 0; i < 8; i = i + 1) begin
        ps2data = keycode[i];
        odd_parity = odd_parity ^ keycode[i];
        ps2clk = 0;
        #100;
        ps2clk = 1;
        #100;
    end

    // Parity bit (odd parity)
    ps2data = odd_parity;
    ps2clk = 0;
    #100;
    ps2clk = 1;
    #100;

    // Stop bit (1)
    ps2data = 1;
    ps2clk = 0;
    #100;
    ps2clk = 1;
    #100;
end
endtask
```

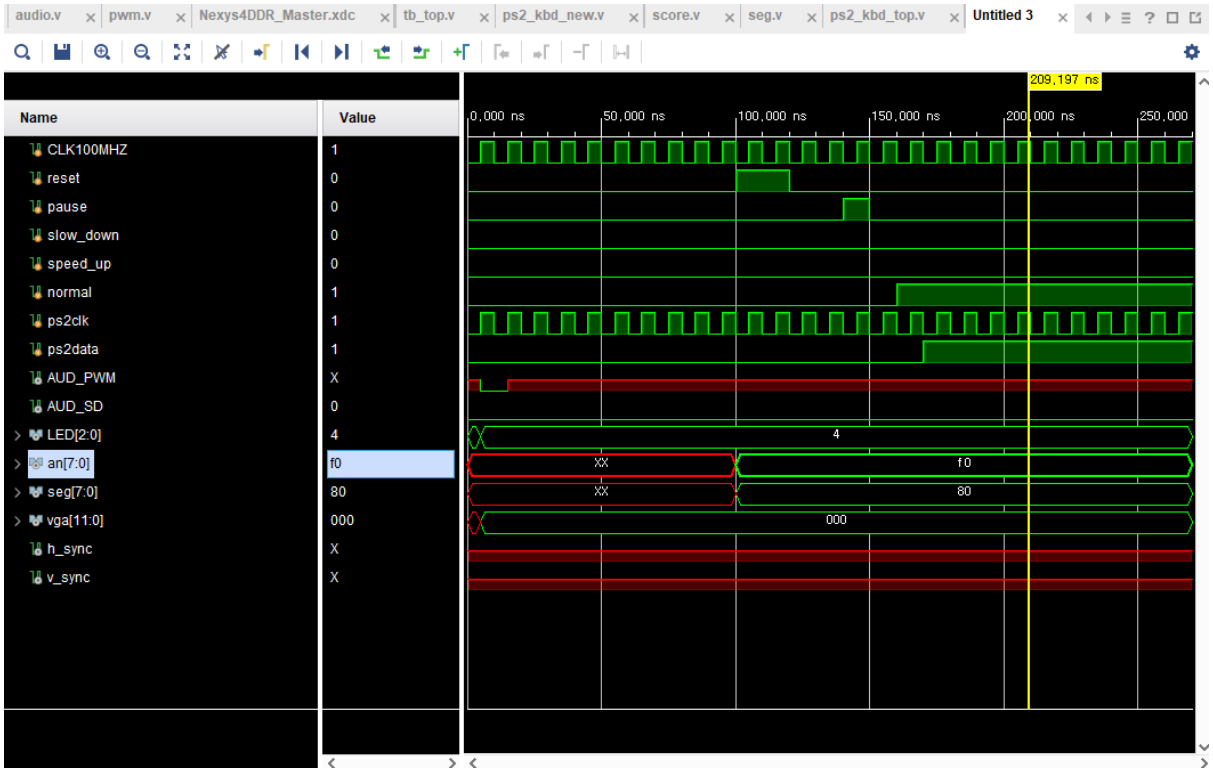
<시뮬레이션 코드 – tb_top.v>

테스트 벤치 – 해당 코드는 tb_top.v 코드로 합성 전 시뮬레이션 결과를 확인하기 위한 코드를 작성한 부분에 가장 중요한 부분을 담은 것입니다.

먼저 top module 을 불러오고 연결해주는 부분은 생략했습니다. 그 이후 시뮬레이션을 실행하기 위해 모든 값을 reset 해주는 것을 설정하고, #100 후에 reset 을 0 으로 바꾼 후 다시 #100 이후 reset 을 해주어 모든 값을 초기화 했습니다.

Ps2_key 값을 컨트롤 해주기 위해 task 를 만들고 그 task 를 활용하여 반복적으로 수행할 수 있는 코드 블록을 정의했습니다.

Ps2 통신을 확인하기 위해서 시작 비트와 데이터 비트, 패리티 비트, 정지비트를 모두 확인하고 태스크 종료를 할 수 있도록 했습니다. 키코드를 전송할 때, ps/2 인터페이스의 전기적 및 타이밍 요구사항을 위해서 준수하며 구현했습니다.



<시뮬레이션 결과>

Reset 신호가 켜지기 전까지는 input 도 없고 제대로 된 신호가 없어 빨간 신호로 알 수 없는 신호들이 나옵니다. 그리고 VGA 같은 경우는 시뮬레이션으로 보기 어려운 신호여서 다른 신호와 다르게 계속해서 알 수 없는 신호로 나옵니다. 해당 결과를 보면, ps2 통신과 seg 가 잘 동작하고 있는 것으로 나오고 PWM 또한 정상적으로 작동하고 있음을 알 수 있습니다.

[4] 합성결과

1. 칩 면적 사용

Nexys a7 칩의 면적 사용률에 대한 분석 결과를 요약합니다. 해당 부분을 통해 주요 구성 요소별 사용률을 검토하여 칩의 리소스가 어떻게 할당되었는지, 할당이 칩 설계 목표와 어떻게 일치하는지를 평가합니다.

(1) 칩 면적 사용율

칩 면적 사용률이 높은 것은 BRAM이 24%, IO가 20%, MMCM이 17%를 차지하고 있음을 사진을 통해 확인할 수 있습니다. 가장 높은 3가지를 중점을 두고 설명하도록 하겠습니다.

BRAM (Block RAM)

해당 설계에서 BRAM이 높은 이유는 BRAM의 특성 때문입니다. BRAM은 주로 대용량 데이터 버퍼링, 데이터 집약적 처리, 파이프라인 처리, 멀티태스킹 또는 병렬 처리 또는 FIFO, 캐시, 큐 등 특수한 데이터 구조가 필요한 경우 높은 사용률이 요구됩니다. 해당 설계에서는 이미지 처리와 비디오 처리부분을 맡고 있는 VGA 모듈을 사용하고 있고, Block mem을 이용하여 coe 파일로 번환된 이미지를 불러와 비디오 처리를 하고 있는 display 모듈이 있기에 높은 사용률을 요구합니다.

IO (Input/Output)

해당 설계에서 IO 부분이 높은 이유는 많은 IO를 사용하고 있기 때문입니다. VGA, Keyboard, Speaker (PWM), Switch, Button, 등 다양한 input과 output 단자가 있습니다. 칩이 다양한 외부 자치와의 통신 능력이 중요하시 하고 있는 것을 보여줍니다. 설계에서는 다중 입출력을 필요로하고 있고, 설계에 맞게 잘 활용되어 가고 있는 것을 확인할 수 있습니다.

MMCM (Mixed-Mode Clock Manager)

MMCM은 칩 설계에서 시계 관리와 가장 연관이 있습니다. 다양한 클록 속도의 필요성과

정밀한 클록 조정이 요구되는 디지털 시스템에서 중요한 역할을 하고 있습니다. 해당 설계에서도 디스플레이 출력을 위해, 해당 해상도에 맞게 출력을 위해 정밀한 클록 조정이 요구되었습니다. 이 과정에서 MMCM 칩 사용률이 올라갔고 중요하다는 것을 보여줍니다.

(2) 칩 면적 분석

해당 설계는 데이터 관리와 다중 입출력 인터페이스, 정교한 시계 관리 기능에 중점을 두고 설계되었음을 알 수 있습니다. 그러나 MMCM은 현재 과도하게 높은 것으로 판단됩니다. 이것은 타이밍 에러가 나타났기 때문입니다. 이점을 제외하고는 Snake Game을 위한 특정 유형의 애플리케이션을 실행할 수 있도록 매우 적합하게 설계되었으며, 성능과 자원 사용의 균형을 효과적으로 맞추고 있음을 보여줍니다. 설계 목표를 반영한 면적 사용률과 설계 전략을 통해 설계 요구 사항을 충족시키는데 필요한 주요 리소스에 대해 높은 성능을 제공할 수 있습니다. 추가적으로 MMCM 사용률을 줄일 수 있도록, 타이밍 컨트롤에 신경을 쓰게 된다면 향후 연구 및 개발 방향도 틀림없이 좋은 결과를 나타낼 수 있음으로 판단됩니다.

2. 클럭 속도

(1) 최대 클럭 속도

분석 결과, 현재 합성된 ASIC 칩은 최대 43.4MHz의 클럭 속도에 도달할 수 있음을 확인했습니다. 이는 Worst Negative Slack (WNS)이 -13.057ns로 측정되었기 때문입니다. WNS는 설계의 타이밍 요구 사항을 충족하지 못하는 최악의 데이터 경로의 타이밍 여유를 나타내며, 이 경우, 클럭 주기를 더 늘려야 할 필요가 있음을 의미합니다.

(2) 속도 제한 요인

타이밍 분석 결과에 따르면, 다음과 같은 요인들이 클럭 속도 제한에 기여하고 있습니다:

- **타이밍 오류**: 디스플레이 클럭의 경우, 100MHz에서 148.5MHz로의 변동 시도 중에 타이밍 오류가 발견되었습니다. 이는 화면 계산 도중 예상치 못한 곱셈 연산으로 인한 추가적인 타이밍 지연을 야기했습니다.

- **로직 복잡성**: 복잡한 논리 연산, 특히 다수의 병렬 연산(|| 연산자 사용)이 한 라인에

서 수행될 때, 타이밍 문제가 발생했습니다. 이는 데이터 경로의 복잡성과 길이 증가로 인해 타이밍 여유가 감소하는 원인이 되었습니다.

(3) 속도 향상 전략

이러한 문제를 해결하고 클럭 속도를 향상시키기 위한 다음 전략을 제안합니다:

- 회로 최적화: 곱셈 연산을 피하고, 가능한 shift 연산으로 대체하여 타이밍을 개선합니

- 로직 간소화: 복잡한 논리 연산을 분할하거나 간소화하여, 클록 사이클 내에서 연산이 완료될 수 있도록 합니다.

이상의 전략을 통해, 칩의 클럭 속도를 현재보다 높은 안정적인 주파수로 조정할 수 있을 것으로 기대됩니다. 이를 통해 성능은 향상되고, 열 및 전력 소비 문제는 최소화될 수 있습니다. 향후 설계 개선 작업에서 이러한 전략들이 적극적으로 고려되어야 할 것입니다.

[5] 결론

Snake 게임 프로그램은 정상적으로 잘 작동합니다. 그래서 코드의 최적화를 위해 코드를 수정하여 실행 속도, 자원 사용량을 효율적으로 하는 것이 향후 응용 가능성 부분에서 가장 좋은 방법이라고 생각했습니다.

1. display모듈

a. display모듈의 91번째,142번째 line인 address<=(y_counter-180)*1320+x_counter-295; 을 * 연산자가 아닌 **shift** 연산자를 이용하여 address를 계산하도록 한다면, 속도 측면에서 두드러진 차이를 볼 수 있습니다.

Ex) address <= ((y_counter-180) << 10) + ((y_counter-180) << 8) + x_counter - 295;

b. snake의 꼬리 부분의 위치를 확인하여 current_x, current_y와 일치하는지 검사하는 로직을 or(||)을 이용해서 105-135번째 총 30라인을 차지하고 있는데, 반복되는 구조를 간소화하기 위하여 반복문을 사용한다면 보다 더 효율적인 코드를 작성할 수 있습니다.

```
else if
(
(current_x == snake_x[1] && current_y == snake_y[1] && snake_piece_is_display[1]==1) ||
(current_x == snake_x[2] && current_y == snake_y[2] && snake_piece_is_display[2]==1) ||
(current_x == snake_x[3] && current_y == snake_y[3] && snake_piece_is_display[3]==1) ||
(current_x == snake_x[4] && current_y == snake_y[4] && snake_piece_is_display[4]==1) ||
(current_x == snake_x[5] && current_y == snake_y[5] && snake_piece_is_display[5]==1) ||
(current_x == snake_x[6] && current_y == snake_y[6] && snake_piece_is_display[6]==1) ||
(current_x == snake_x[7] && current_y == snake_y[7] && snake_piece_is_display[7]==1) ||
(current_x == snake_x[8] && current_y == snake_y[8] && snake_piece_is_display[8]==1) ||
(current_x == snake_x[9] && current_y == snake_y[9] && snake_piece_is_display[9]==1) ||
(current_x == snake_x[10] && current_y == snake_y[10] && snake_piece_is_display[10]==1) ||
(current_x == snake_x[11] && current_y == snake_y[11] && snake_piece_is_display[11]==1) ||
(current_x == snake_x[12] && current_y == snake_y[12] && snake_piece_is_display[12]==1) ||
(current_x == snake_x[13] && current_y == snake_y[13] && snake_piece_is_display[13]==1) ||
(current_x == snake_x[14] && current_y == snake_y[14] && snake_piece_is_display[14]==1) ||
(current_x == snake_x[15] && current_y == snake_y[15] && snake_piece_is_display[15]==1) ||
(current_x == snake_x[16] && current_y == snake_y[16] && snake_piece_is_display[16]==1) ||
(current_x == snake_x[17] && current_y == snake_y[17] && snake_piece_is_display[17]==1) ||
(current_x == snake_x[18] && current_y == snake_y[18] && snake_piece_is_display[18]==1) ||
(current_x == snake_x[19] && current_y == snake_y[19] && snake_piece_is_display[19]==1) ||
(current_x == snake_x[20] && current_y == snake_y[20] && snake_piece_is_display[20]==1) ||
(current_x == snake_x[21] && current_y == snake_y[21] && snake_piece_is_display[21]==1) ||
(current_x == snake_x[22] && current_y == snake_y[22] && snake_piece_is_display[22]==1) ||
(current_x == snake_x[23] && current_y == snake_y[23] && snake_piece_is_display[23]==1) ||
(current_x == snake_x[24] && current_y == snake_y[24] && snake_piece_is_display[24]==1) ||
(current_x == snake_x[25] && current_y == snake_y[25] && snake_piece_is_display[25]==1) ||
(current_x == snake_x[26] && current_y == snake_y[26] && snake_piece_is_display[26]==1) ||
(current_x == snake_x[27] && current_y == snake_y[27] && snake_piece_is_display[27]==1) ||
(current_x == snake_x[28] && current_y == snake_y[28] && snake_piece_is_display[28]==1) ||
(current_x == snake_x[29] && current_y == snake_y[29] && snake_piece_is_display[29]==1) ||
(current_x == snake_x[30] && current_y == snake_y[30] && snake_piece_is_display[30]==1) ||
(current_x == snake_x[31] && current_y == snake_y[31] && snake_piece_is_display[31]==1)
)
vga<=12'b0000_1000_0010;
```

2. snake 모듈

a. display 모듈에서와 비슷하게 snake가 자기 자신에게 부딪히는 조건을 검사하는 로직을 or(||)을 이용해서 93-123번째 총 30라인을 차지하고 있는데, 반복되는 구조를 간소화하기 위하여 반복문을 사용한다면 보다 더 효율적인 코드를 작성할 수 있습니다.

```
if (
(snake_x[0]==snake_x[1] && snake_y[0]==snake_y[1] && snake_piece_is_display[1]==1) ||
(snake_x[0]==snake_x[2] && snake_y[0]==snake_y[2] && snake_piece_is_display[2]==1) ||
(snake_x[0]==snake_x[3] && snake_y[0]==snake_y[3] && snake_piece_is_display[3]==1) ||
(snake_x[0]==snake_x[4] && snake_y[0]==snake_y[4] && snake_piece_is_display[4]==1) ||
(snake_x[0]==snake_x[5] && snake_y[0]==snake_y[5] && snake_piece_is_display[5]==1) ||
(snake_x[0]==snake_x[6] && snake_y[0]==snake_y[6] && snake_piece_is_display[6]==1) ||
(snake_x[0]==snake_x[7] && snake_y[0]==snake_y[7] && snake_piece_is_display[7]==1) ||
(snake_x[0]==snake_x[8] && snake_y[0]==snake_y[8] && snake_piece_is_display[8]==1) ||
(snake_x[0]==snake_x[9] && snake_y[0]==snake_y[9] && snake_piece_is_display[9]==1) ||
(snake_x[0]==snake_x[10] && snake_y[0]==snake_y[10] && snake_piece_is_display[10]==1) ||
(snake_x[0]==snake_x[11] && snake_y[0]==snake_y[11] && snake_piece_is_display[11]==1) ||
(snake_x[0]==snake_x[12] && snake_y[0]==snake_y[12] && snake_piece_is_display[12]==1) ||
(snake_x[0]==snake_x[13] && snake_y[0]==snake_y[13] && snake_piece_is_display[13]==1) ||
(snake_x[0]==snake_x[14] && snake_y[0]==snake_y[14] && snake_piece_is_display[14]==1) ||
(snake_x[0]==snake_x[15] && snake_y[0]==snake_y[15] && snake_piece_is_display[15]==1) ||
(snake_x[0]==snake_x[16] && snake_y[0]==snake_y[16] && snake_piece_is_display[16]==1) ||
(snake_x[0]==snake_x[17] && snake_y[0]==snake_y[17] && snake_piece_is_display[17]==1) ||
(snake_x[0]==snake_x[18] && snake_y[0]==snake_y[18] && snake_piece_is_display[18]==1) ||
(snake_x[0]==snake_x[19] && snake_y[0]==snake_y[19] && snake_piece_is_display[19]==1) ||
(snake_x[0]==snake_x[20] && snake_y[0]==snake_y[20] && snake_piece_is_display[20]==1) ||
(snake_x[0]==snake_x[21] && snake_y[0]==snake_y[21] && snake_piece_is_display[21]==1) ||
(snake_x[0]==snake_x[22] && snake_y[0]==snake_y[22] && snake_piece_is_display[22]==1) ||
(snake_x[0]==snake_x[23] && snake_y[0]==snake_y[23] && snake_piece_is_display[23]==1) ||
(snake_x[0]==snake_x[24] && snake_y[0]==snake_y[24] && snake_piece_is_display[24]==1) ||
(snake_x[0]==snake_x[25] && snake_y[0]==snake_y[25] && snake_piece_is_display[25]==1) ||
(snake_x[0]==snake_x[26] && snake_y[0]==snake_y[26] && snake_piece_is_display[26]==1) ||
(snake_x[0]==snake_x[27] && snake_y[0]==snake_y[27] && snake_piece_is_display[27]==1) ||
(snake_x[0]==snake_x[28] && snake_y[0]==snake_y[28] && snake_piece_is_display[28]==1) ||
(snake_x[0]==snake_x[29] && snake_y[0]==snake_y[29] && snake_piece_is_display[29]==1) ||
(snake_x[0]==snake_x[30] && snake_y[0]==snake_y[30] && snake_piece_is_display[30]==1) ||
(snake_x[0]==snake_x[31] && snake_y[0]==snake_y[31] && snake_piece_is_display[31]==1)
)
hit_itself<=1;
```

b. snake 모듈 내에서 스네이크의 각 부분이 움직이는 로직을 나타내는 부분이 있습니다(179-270번째 라인) 이 부분 또한 각 스네이크 조각의 위치를 개별적으로 갱신하고 있는데, 이러한 반복되는 구조를 반복문을 사용하여 간소화 할 수 있습니다.

Ex)

verilog Copy code

```
integer i;
for (i = 31; i > 0; i = i - 1) begin
    snake_x[i] <= snake_x[i - 1];
    snake_y[i] <= snake_y[i - 1];
end
```

이러한 최적화 과정은 Snake 게임 프로그램의 실행 속도를 빠르게 하고 자원 사용량을 현저히 줄이는 데 기여할 것입니다. 하드웨어 설계의 맥락에서, 효율적인 로직 구조는 처리 속도의 개선, 반응 시간의 단축, 그리고 시스템 전체의 처리량 증대로 이어질 수 있습니다. 더 나아가, 최적화는 클럭 속도에 대한 요구 사항을 낮추고 전력 소모를 줄이며, 전체 시스템의 안정성과 신뢰성을 향상시키는 중요한 수단입니다. 따라서, 코드 최적화는 단순히 성능 향상을 넘어 시스템 설계의 근본적인 질을 높이는 핵심적인 접근 방식입니다.