SNAKE & LADDER PROJECT

PRESS START TO BEGIN

Designed by Team 5

PRESENTER : Chae Minseok

이동관 임용진 정예찬 채민석

CONTENTS

PROJECT OVERVIEW

PROLOGUE

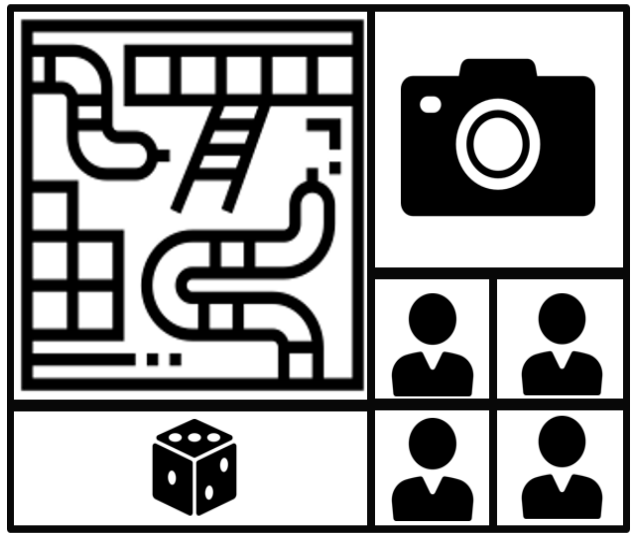
INTRODUCTION

PROLOGUE : INTRODUCTION

프로젝트 목표

- OV7670 카메라, VGA, 게임 로직을 단일 FPGA에 통합하여 개구리 말을 사용하는 뱀사다리 게임 시스템 구현
- 디지털 회로, 영상 처리, 임베디드 시스템을 아우르는 통합 하드웨어 설계 역량 향상

구현 목표

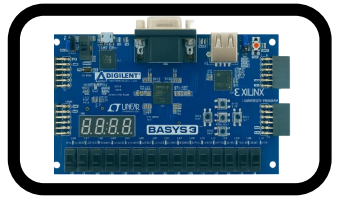


< VGA Design>

개발 환경

< System Verilog >

<Vivado>



<Basys3>

<Buzzer>

<OV7670>

PROLOGUE : INTRODUCTION

HOW to PLAY?

인원 수 선택
- 최대 4인 까지 설정 가능

카메라 촬영
- 플레이어는 게임 시작전 카메라 촬영

주사위 굴리기 !
- 주사위를 통해 자신의 말을 이동

승리 조건
- 가장 먼저 49번 위치에 도달 하는 사람이 승리!

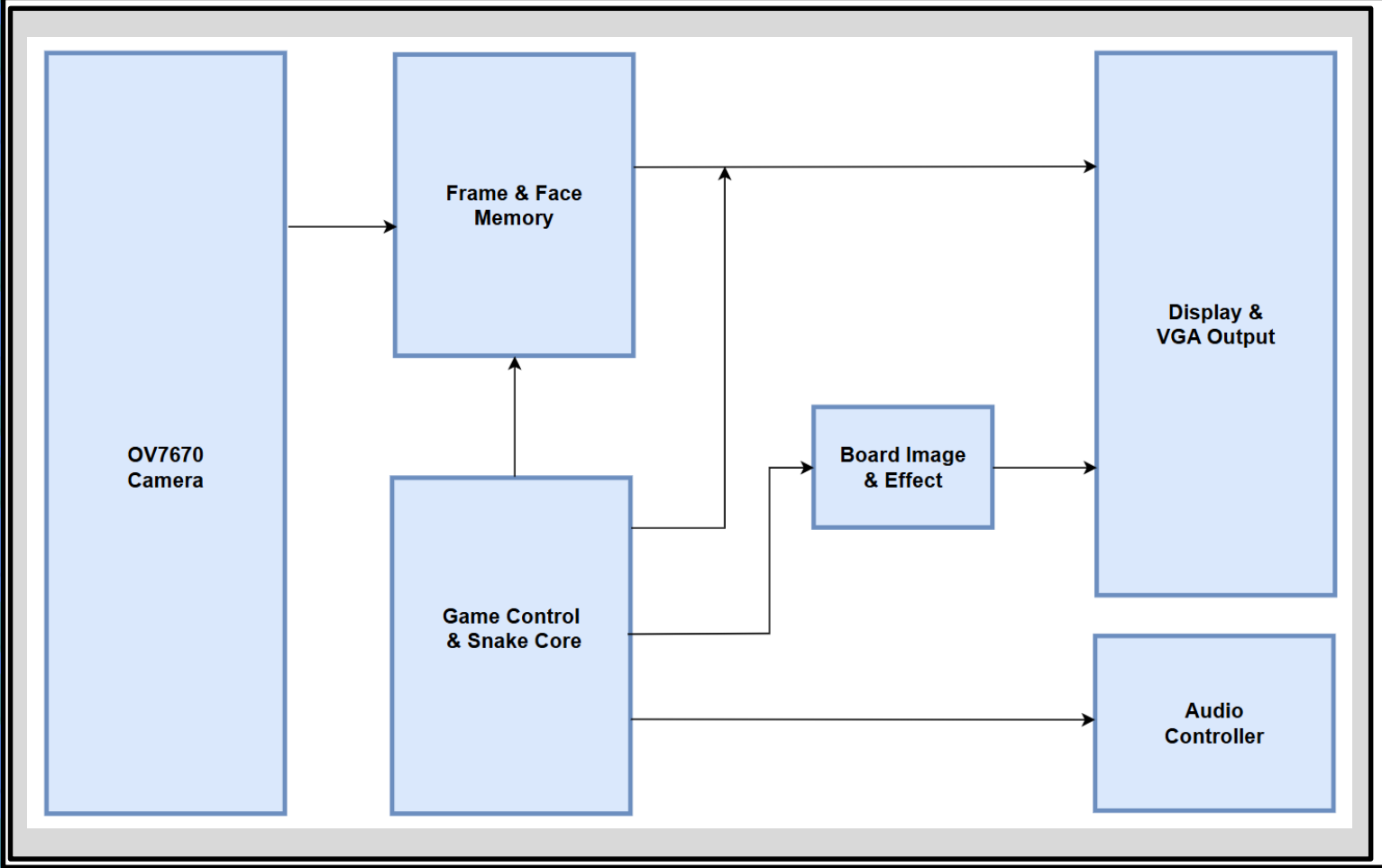
특별 규칙

와 테초마을이야
- 앞서 가던 플레이어를 시작지점을 보냅니다

야생에 뱀이 나타났다!
- 뱀에 물렸습니다. 뱀의 꼬리로 이동합니다

황금 사다리
- 승리로 가는 지름길! 사다리 끝으로 이동합니다

PROLOGUE : INTRODUCTION



OV7670 Camera
- 카메라 설정 및 영상 캡처

Frame & Face Memory
- 메인 화면 · 얼굴 이미지를 BRAM에 읽기/쓰기

Game Control & Snake Core
- 버튼 기반 턴, 주사위, 말 이동 · 이벤트 계산

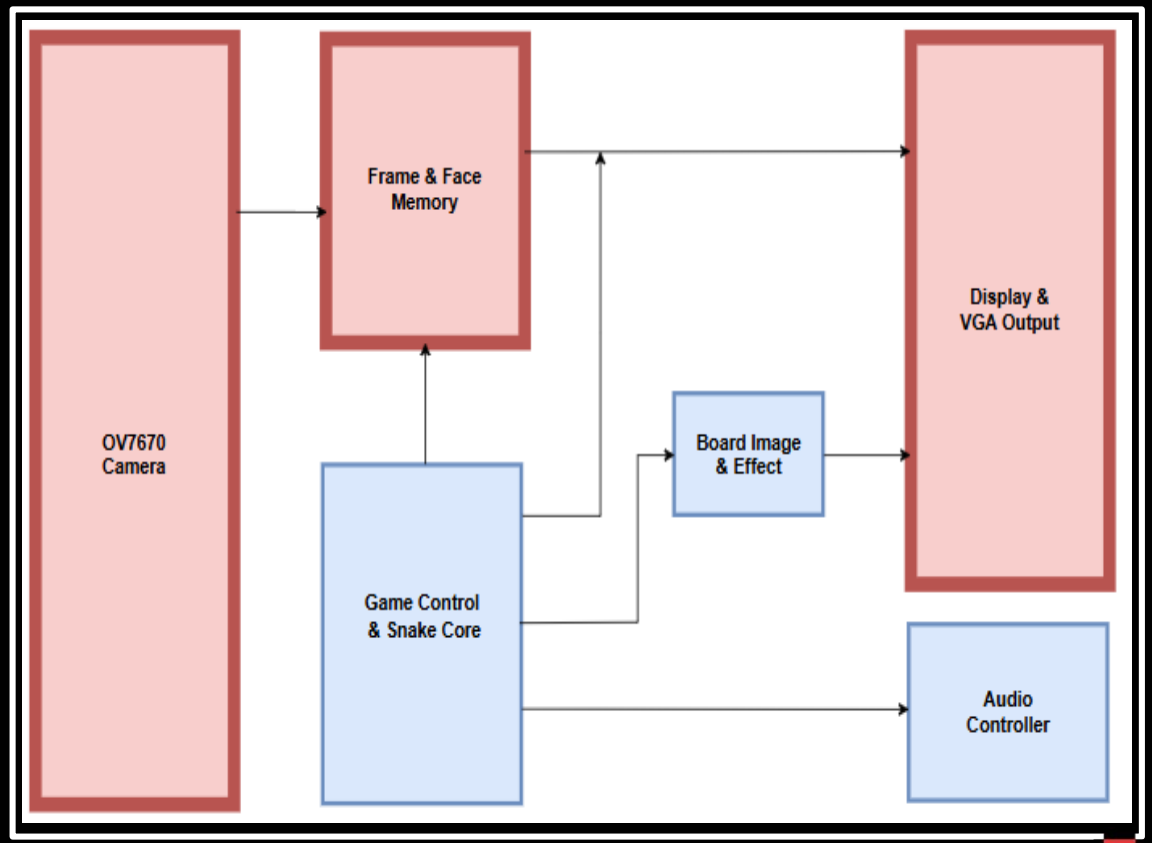
Board Image & Effect(Shake)
- 보드 이미지 생성 및 흔들림 효과 적용

Display & VGA Out
- 보드 · 카메라 · 말을 합성해 VGA 신호 출력

Audio Controller
- 게임 이벤트에 맞춰 효과음 PWM 출력

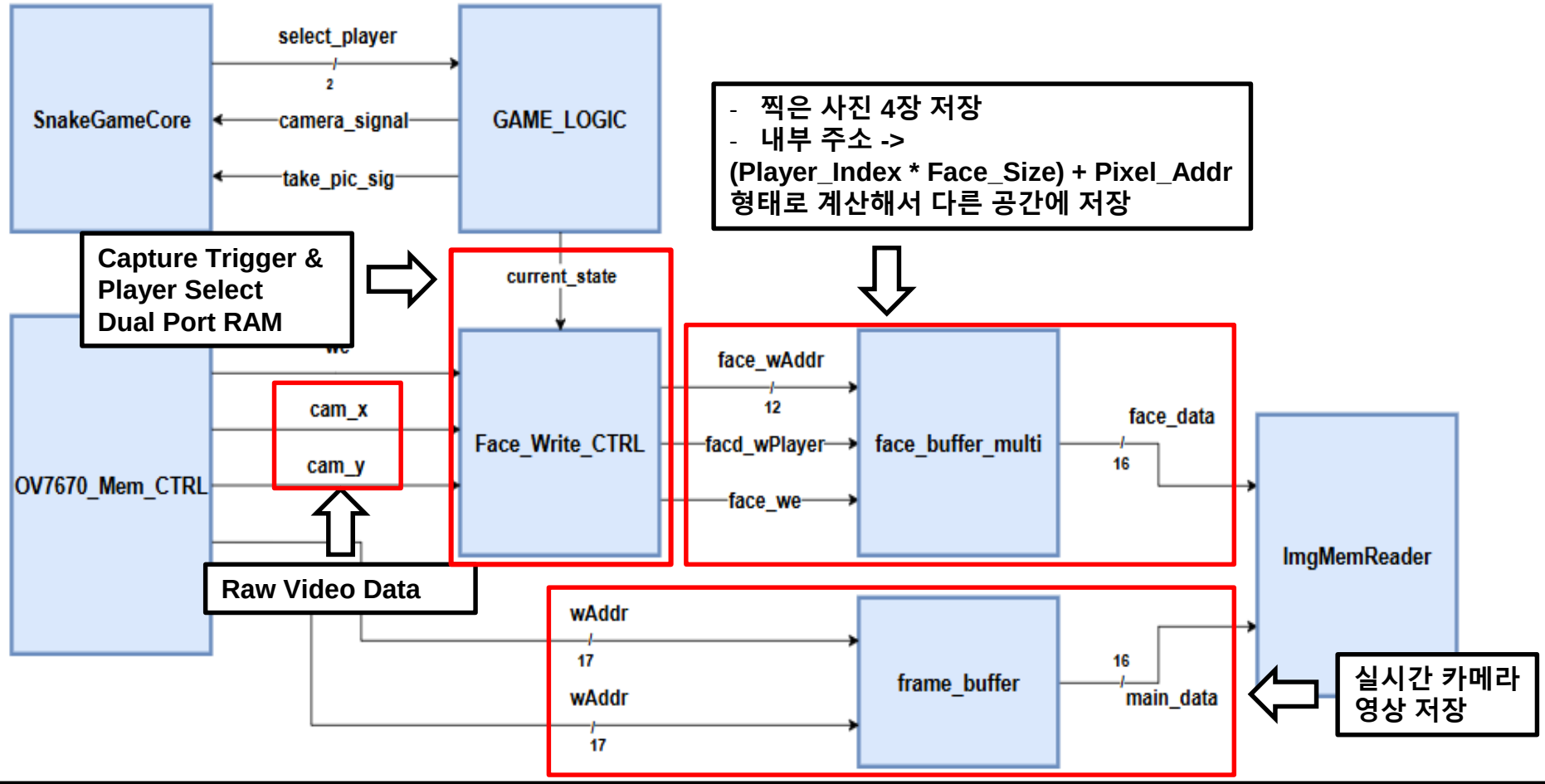
STAGE 1

CHOOSE YOUR PLAYER



STAGE 1 : How To Select Player?

Camera Logic BlockDiagram



- 찍은 사진 4장 저장
- 내부 주소 ->
(Player_Index * Face_Size) + Pixel_Addr
형태로 계산해서 다른 공간에 저장

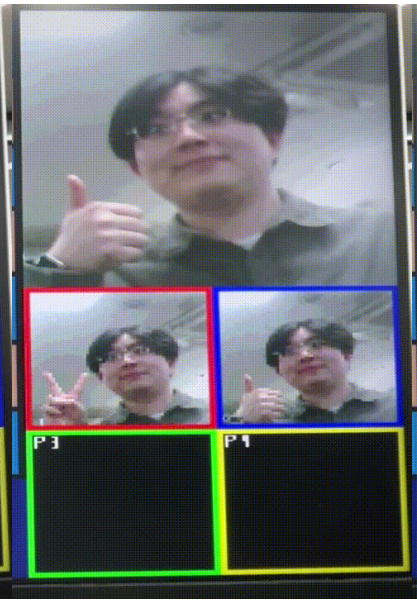
STAGE 1 : How To Select Player?

Player 인원 수 설정 시연 영상

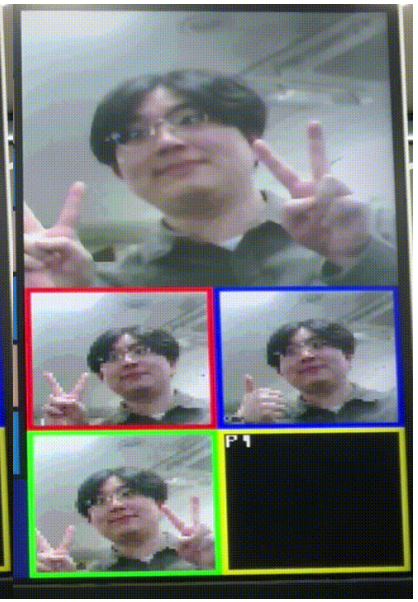
< Player 수 설정 화면>

< Player 1>

< Player 2>



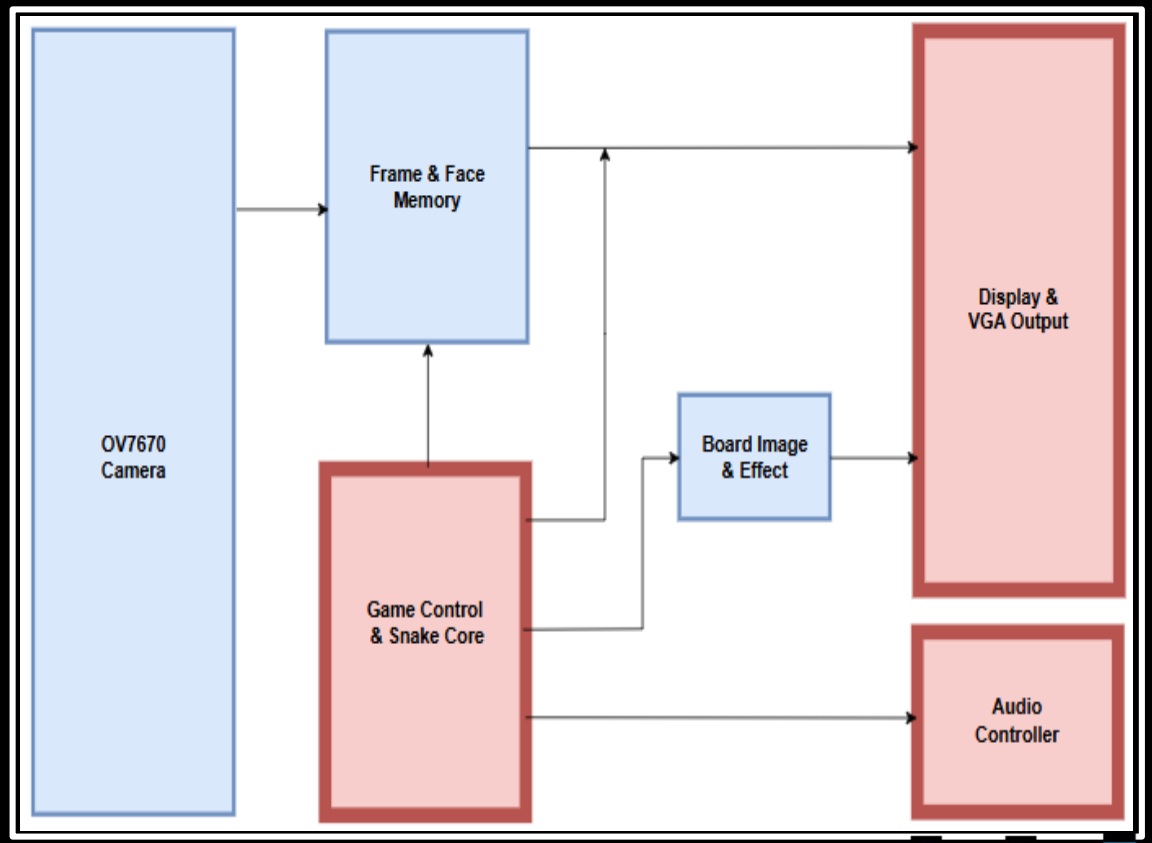
< Player 3>



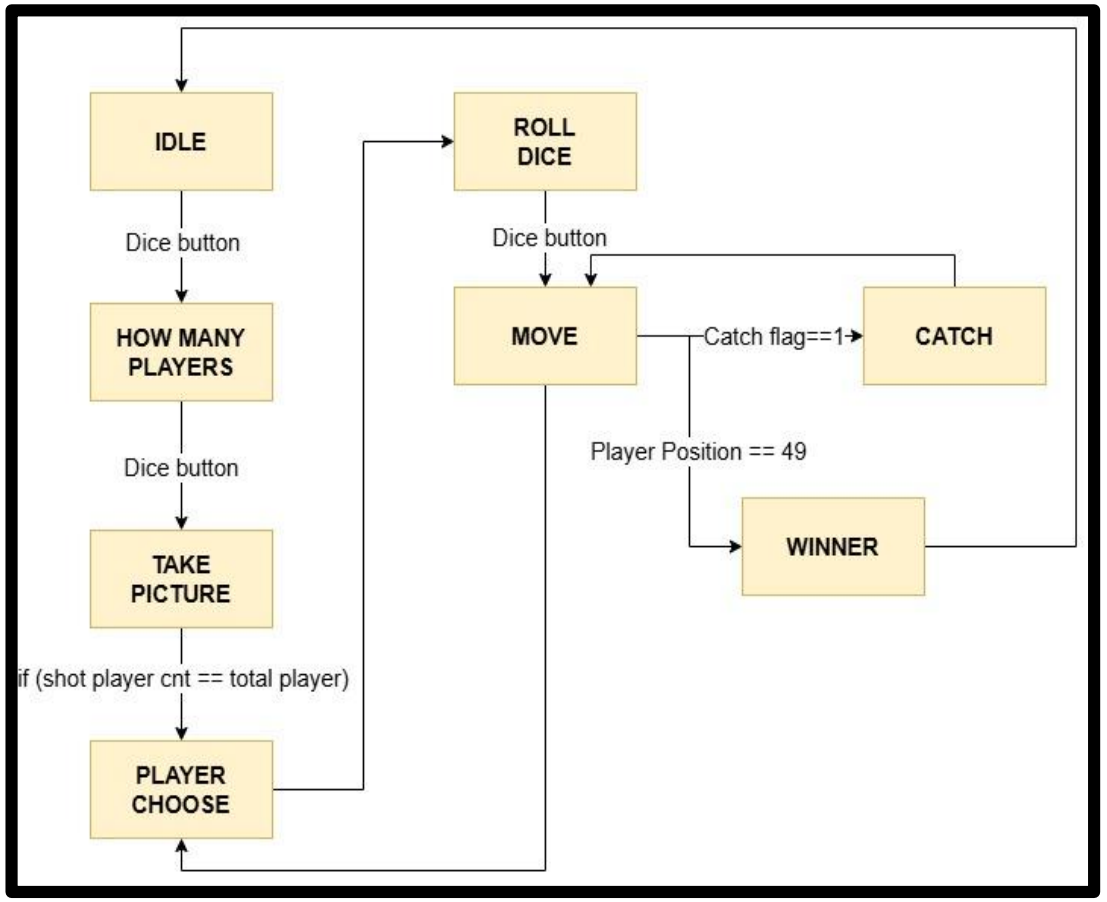
< Player 4>

STAGE 2

THE RULES OF GAME



STAGE 2 : THE RULES OF THE GAME



PLAYER CHOOSE

플레이어 턴
현재 플레이어 턴을 확인

ROLL DICE

주사위 굴리기
1~6까지 Step by Step 이동

MOVE

말 이동 (뱀, 사다리, 잡기, 승자 판정)
사다리 : 앞으로, 뱀 : 뒤로

CATCH

잡기
잡힌 상대를 시작 지점으로 이동

WINNER

승리
49번 칸에 먼저 도착한 사람이 승리!

STAGE 2 : THE RULES OF THE GAME

PLAYER CHOOSE & DICE ROLL

오른쪽 버튼을 눌러
주사위 굴리기 진행

플레이어 턴 판단

```
PLAYER_CHOOSE: begin
    gaming_now_next = 1'b1;
    if (player_phase > player_cnt) player_phase_next = 2'd0;
    state_next = ROLL_DICE;
end
```

플레이어 인원 수 & 플레이어 차례 카운터 비교로 턴 판단

주사위 난수

```
always_comb begin
    lfsr_next = {lfsr[6:0], lfsr[7] ^ lfsr[5] ^ lfsr[4] ^ lfsr[3]};
end
```

LFSR(Linear Feedback Shift Register) 난수 생성기

$x^8 + x^6 + x^5 + x^4 + 1$ 로 255번의 서로 다른 난수 패턴

dice_result = (lfsr % 6) + 1 로 주사위 값 결정

STAGE 2 : THE RULES OF THE GAME

MOVE

주사위 굴려 이동 이동 후
도착 칸에 대해 이벤트 발생

뱀 칸 도착 시

뱀 머리 칸 도달 시
뱀 꼬리 칸으로 이동

```
function automatic [5:0] warp_pos(input [5:0] p);
    case (p)
        // Ladder
        6'd5: warp_pos = 6'd18;
        6'd16: warp_pos = 6'd31;
        6'd39: warp_pos = 6'd45;
        // Snake
        6'd20: warp_pos = 6'd7;
        6'd30: warp_pos = 6'd15;
        6'd47: warp_pos = 6'd25;
        default: warp_pos = p;
    endcase
endfunction
```

사다리 칸 도착 시

사다리 밑 칸 도달 시
사다리 위 칸으로 이동

STAGE 2 : THE RULES OF THE GAME

CATCH

상대방을 잡을 시

```
for (int i = 0; i < 4; i++) begin
    if (i != cur_p && i <= player_cnt) begin
        if (pos_reg[i] == new_pos) begin
            catch_flag          = 1'b1;
            caught_player_next = i[1:0];
        end
    end
end
```

저장된 플레이어 위치 정보 기반으로 잡힘 판단

```
pos_next[caught_player_reg] = 6'd1;
```

잡힌 플레이어는 1번 칸으로 이동

```
state_next = ROLL_DICE;
```

잡은 플레이어는 1번 더 주사위 굴리기

STAGE 2 : THE RULES OF THE GAME

WINNER

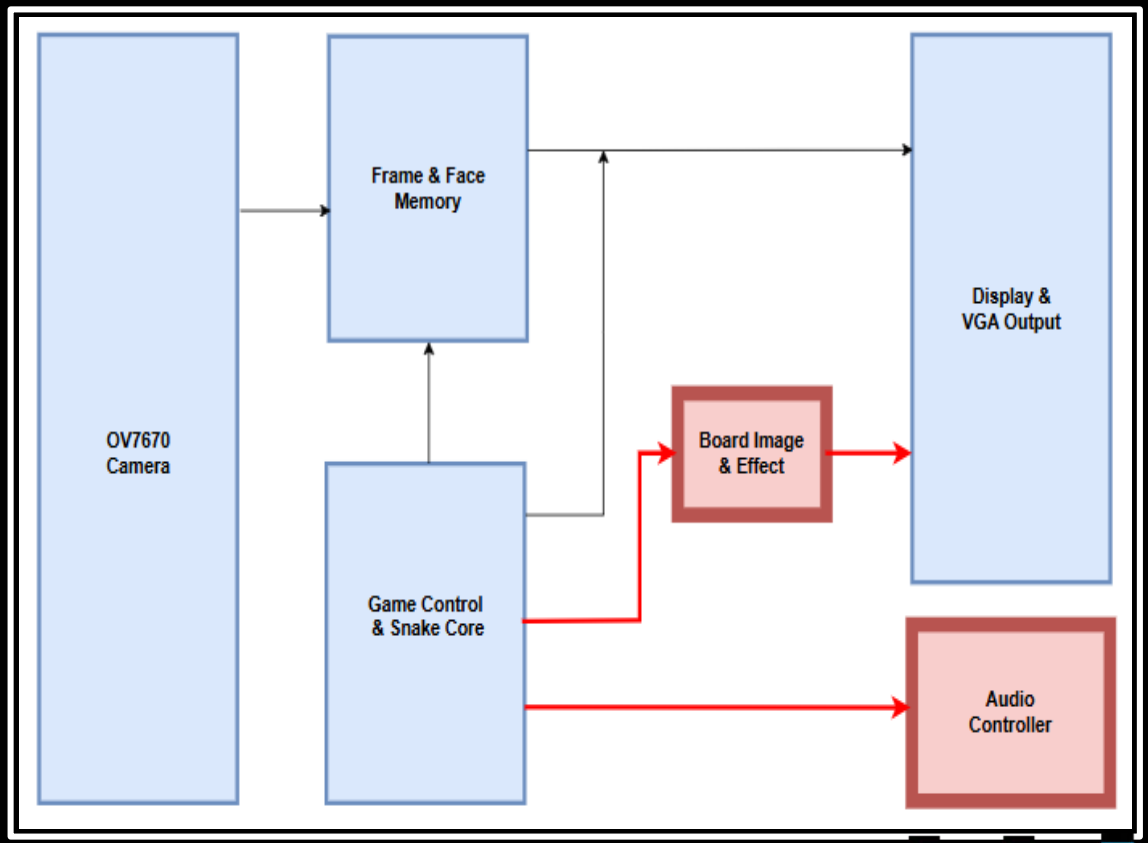
승자 판단

```
if (new_pos == 6'd49) begin
    winner_next = cur_p;
    state_next  = WINNER;
end else begin
```

플레이어가 이동하며 위치가 49번 칸이면
승자 판정 및 게임종료

STAGE 3

GAME EFFECTS



STAGE 3 : GAME EFFECTS

Shake Hit Effect

말이 뱀의 주소에 위치하면 board가 흔들리는 방식

```
// 기본은 원위치
dx = 11'sd0;
dy = 11'sd0;
case (phase)
    3'd0: begin dx =  11'sd10; dy =  11'sd0;   end
    3'd1: begin dx = -11'sd20; dy =  11'sd5;   end
    3'd2: begin dx = -11'sd15; dy = -11'sd20;  end
    3'd3: begin dx =  11'sd10; dy =  11'sd13;  end
    3'd4: begin dx =  11'sd0;  dy =  11'sd0;   end
    default: ;
endcase
```

총 5개의 상태를 순차적으로 진행하여 보드의 주소가 이동하며 흔들리는 것 처럼 표현

STAGE 3 : GAME EFFECTS

Catch Effect

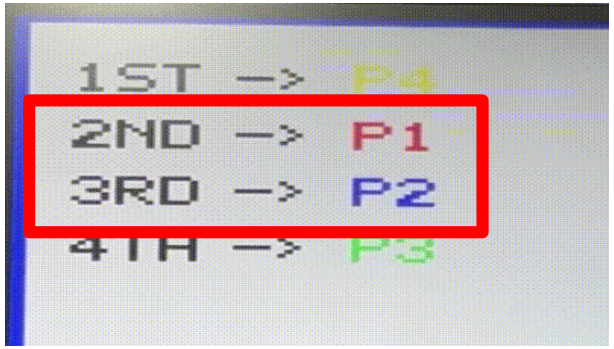
```
if (catch_flag) begin
    state_next = CATCH;
    caught_status_next[caught_player_next] = 1'b1;
    catch_event_pulse = 1'b1;
end else begin
    if (player_phase == player_cnt)
        player_phase_next = 2'd0;
    else player_phase_next = player_phase + 2'd1;

    state_next = PLAYER_CHOOSE;
end
```

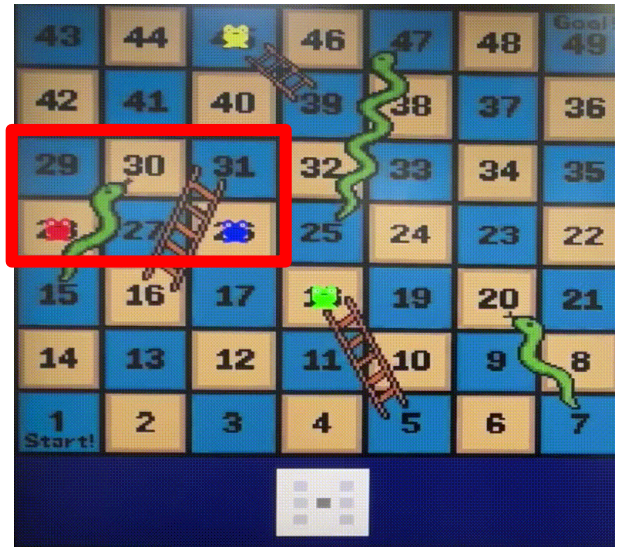
Game Core Module에서
Catch_flag가 발생하면
Caught_status 값이 1발생

```
if (caught_status[face_player_idx]) begin
    {r_port, g_port, b_port} = {
        gray_val,      // Red (Bright)
        gray_val >> 1, // Green (Dark)
        gray_val >> 1  // Blue (Dark)
    };
```

Game Core Module과 연결된
Img Reader에
Caught_Status값이 들어오면
RED 값만 올려 붉은 EFFECT
표시

STAGE 3 : GAME EFFECTS





Real Time Rank Update

```
always_comb begin
    pos[0] = pos_P1;
    pos[1] = pos_P2;
    pos[2] = pos_P3;
    pos[3] = pos_P4;

    for (i = 0; i < 4; i = i + 1)
        idx[i] = i[1:0];  // [1:0]으로 형변환

    for (k = 0; k < 4; k = k + 1) begin
        max_k = k;
        for (j = k + 1; j < 4; j = j + 1) begin
            if (pos[idx[j]] > pos[idx[max_k]]) max_k = j;
        end
        if (max_k != k) begin
            tmp_idx    = idx[k];
            idx[k]     = idx[max_k];
            idx[max_k] = tmp_idx;
        end
    end

    // 정렬
    rank1_pl = idx[0];
    rank2_pl = idx[1];
    rank3_pl = idx[2];
    rank4_pl = idx[3];
```

각 플레이어 말을 pos[n]에 저장

idx[n]에 플레이어 번호 넣고 선택 정렬

정렬 이 후 rank1 ~ 4 까지

플레이어 번호 저장 및 출력

FINAL STAGE

DEMONSTRATION VIDEO

FINAL STAGE : DEMONSTRATION VIDEO

HIDDEN STAGE

TROUBLE SHOOTING & DISCUSSION

EPILOGUE : TROUBLE SHOOTING

SCCB ↔ OV7670 연결 문제 (Timing 문제)

Before

```
logic [8:0] div_cnt;
logic       I2C_clk_400khz;

always_ff @(posedge clk or posedge reset) begin
    if (reset) begin
        div_cnt        <= 0;
        I2C_clk_400khz <= 0;
    end else if (div_cnt == 9'd249) begin
        div_cnt        <= 0;
        I2C_clk_400khz <= ~I2C_clk_400khz;
    end else begin
        div_cnt <= div_cnt + 1'b1;
    end
end
always_ff @(posedge I2C_clk_400khz or posedge reset) begin
    if (reset) begin
        scl_state <= SCL_IDLE;
        r_scl     <= 1;
```

분주된 clk를 새로운 클럭 으로 사용해서 I2C setup, hold time이 깨지면서 통신이 불안정 함.

After

```
always_ff @(posedge clk) begin
    if (reset) begin
        scl_state    <= SCL_IDLE;
        r_scl        <= 1;
        r_I2C_clk_en <= 0;
    end else begin
        case (scl_state)
            SCL_IDLE:
                if (sda_state == SDA_START) begin
                    r_I2C_clk_en <= 1;
                    scl_state    <= SCL_START;
                end
            SCL_START:
                if (I2C_clk_400khz) begin
                    r_scl     <= 0;
                    scl_state <= SCL_H2L;
                end
            SCL_H2L:
                if (I2C_clk_400khz) begin
                    r_scl     <= 0;
                    scl_state <= SCL_L2L;
                end
            SCL_L2L:
                if (I2C_clk_400khz) begin
                    r_scl     <= 1;
                    scl_state <= SCL_L2H;
                end
        endcase
    end
end
```

모든 플립플롭은 100MHz 메인 클럭에만 동기화하고,
I2C_clk_400khz는 Clock Enable처럼 사용해서
타이밍 마진을 확보하고 통신이 안정화됨.

문제 원인

- 분주된 slow_clk를 로직 클럭으로 사용
- 분주 클럭의 skew/jitter로 I2C 셋업·홀드 타임 깨짐

해결 방법

- 모든 로직을 100MHz 메인 클럭 하나에 동기화
- Clock Enable 신호로만 I2C 타이밍을 제어

EPILOGUE : TROUBLE SHOOTING

SCCB ↔ OV7670 연결 문제 (Protocol 문제)

Before

```
READ_WAIT_ACK: begin
    sda_drive_next_TX = 1'b0;  // 마스터가 SDA를 구동

    if (scl_pos) begin
        if (sda == 1'b0) begin    // ACK
            rd_ptr_next  = rd_ptr + 1;
            bit_cnt_next = 4'd7;
            state_next   = READ_DATA;
        end else begin            // NACK
            rd_ptr_next = rd_ptr + 1;
            state_next  = IDLE;
        end
    end
end
```

<기존 I2C 방법 설계>
ACK가 기대한대로 나오지않으면 바로 IDLE로 떨어지거나 통신을 끊도록 설계 되었으므로 SCCB 규격에서는 통신이 자주 끊기게 됨

After

```
if (scl_state == SCL_H2L && I2C_clk_400khz) begin

    if (bitCount == 9) begin
        bitCount_next = 1;
        r_sda_next    = initData[dataBit];
        dataBit_next  = dataBit - 1;
        sda_state_next = (sda_state == DEVICE_ID)   ? ADDRESS_REG :
                         (sda_state == ADDRESS_REG) ? DATA_REG    :
                                                      SDA_STOP;
        if (sda_state == DATA_REG) begin
            bitCount_next = 0;
            dataBit_next  = 23;
            r_sda_next    = 0;
        end
    end else if (bitCount == 8) begin
        r_sda_next    = 1;
        bitCount_next = bitCount + 1;
    end else begin
        r_sda_next    = initData[dataBit];
        dataBit_next = (dataBit == 0) ? dataBit : dataBit - 1;
        bitCount_next = bitCount + 1;
    end
end
```

< SCCB 전용 CTRL 설계 >
ACK 싸이클을 한 번 쉬어가는 구간으로 처리후 LUT에 있는 레지스터들을 끝까지 보내는 Blind Write 방식으로 변경

문제 원인

- OV7670은 SCCB방식이지만 이전 코드는 표준 I2C처럼 ACK를 엄격하게 검사
- ACK를 한 번 놓치면 FSM이 ERROR 상태로 멈춤

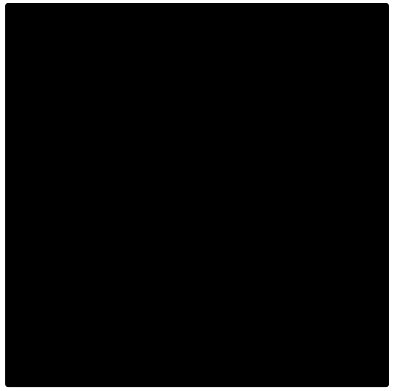
해결 방법

- ACK를 don't care로 처리하는 SCCB 전용 컨트롤러 구현
- Blind Write 방식으로 모든 레지스터를 끝까지 전송
  → 카메라 초기화 안정성 상승

EPILOGUE : TROUBLE SHOOTING

Shaking이 제대로 안나왔던 문제

문제 현상



예시

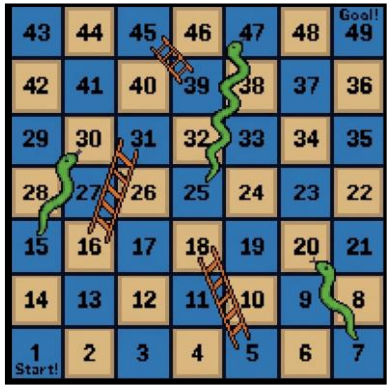
화면 흔들림(shaking) 효과는 보이지만
뱀사다리 보드 배경 이미지는 출력되지 않음

문제 원인

```
always_comb begin
    x_pix_shake = x_pixel + shake_dx;
    y_pix_shake = y_pixel + shake_dy;
```

흔들림 값(좌우/상하 오프셋)을 그대로
화면 좌표에 더한 뒤
이 흔들린 좌표를 그대로 보드 영역 체크
+ 메모리 주소 계산에 동시에 사용했었음

해결 방법



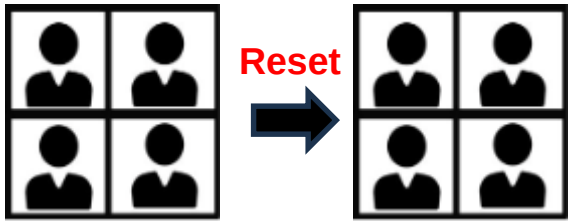
예시

Frame Buffer 안의 보드 좌표는 항상 고정
메모리는 이 고정된 보드 좌표로만 접근하게 변경

EPILOGUE : TROUBLE SHOOTING

BRAM에 저장된 Player사진이 reset 안되는 문제

문제



Take Picture 상태에서
촬영 사진이 정상적으로 저장

But

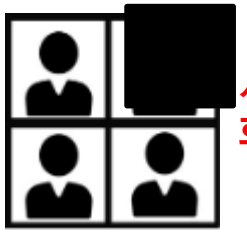
Reset 시 기존 플레이에서
촬영된 사진이 초기화되지 않는 문제

과정

촬영 저장 모듈

1 face_buffer_multi (BRAM) BRAM을 Reset

LUTRAM으로 변경되며 용량 문제 발생

2  사각 박스로 화면 가리기

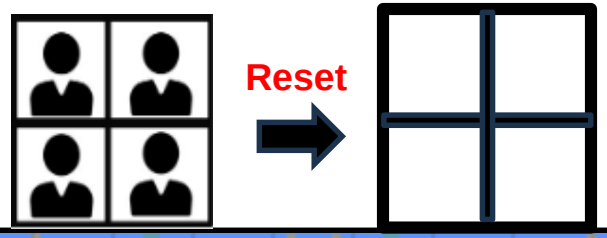
근본적 문제 해결 방식이 아니다.

해결

3

```
always_ff @(posedge pclk) begin
    if (reset_start) begin
        clear_running <= 1;
        clear_ptr     <= 0;
    end else if (clear_running) begin
        if (clear_ptr == 14'd16383) clear_running <= 0;
        else clear_ptr <= clear_ptr + 14'd1;
    end
end
```

Clear 용 FSM을 구현해
Reset 시 RAM 모든 영역에 0을 Write



EPILOGUE : DISCUSSION

[정예찬]

초기에는 기능 구현에 집중했으나, 개발 과정에서 제한된 BRAM 용량과 타이밍 에러에 직면하며 하드웨어 설계 시 리소스 최적화와 신호 동기화가 시스템 안정성에 필수적임을 깊이 깨달았습니다.
실시간 영상 처리를 위한 다운스케일링 로직과 SCCB 제어용 FSM을 직접 설계하면서, 단순한 코드 작성을 넘어 물리적 제약과 정밀한 타이밍을 고려하는 하드웨어적 사고방식을 길렀습니다.
이번 프로젝트는 FPGA 위에서 영상, 로직, 사운드가 결합된 하나의 완성된 시스템(SoC)을 구축해내며 엔지니어로서 큰 성취감을 느낀 계기가 되었습니다.

[채민석]

게임 코어와 연동되는 효과를 구현하는 과정에서, 동일 코어 안에 효과와 상태 변화를 모두 넣어 타이밍을 처리하다 보니 로직이 꼬이고 출력이 불안정해지는 문제가 있었습니다. 이를 해결하기 위해 효과 전용 모듈을 분리하고 타이밍 순서를 재구성해 동작을 안정화했습니다. 또한 게임 로직과 카메라 로직을 각각 구현한 뒤 TOP에서 통합하는 과정에서 sync 불일치로 원하는 화면이 출력되지 않는 문제가 발생했으나, 두 모듈의 입출력 타이밍과 인터페이스를 재정의하여 해결했으며 이를 통해 기능 분할뿐 아니라 최종 통합 단계에서의 협업과 조율의 중요성을 깨달았습니다.

[이동관]

뱀 사다리 게임 자체의 게임 코어 FSM으로 구성했습니다. 화면에 띄우기 위한 한 렌더링 모듈을 제작하며 VGA 역량을 길렀습니다. 저장된 사진 리셋 디버깅 등을 진행하며 문제해결능력을 길렀습니다.
Notion에 코드를 백업하는 방식으로 버전별로 프로젝트 코드를 관리했습니다. 수정사항과 개선 내용을 기록하고 관리하는 과정에서 코드관리의 중요성을 느꼈습니다.
협업 과정을 통해 적절하게 프로젝트 업무량을 분배하고 진행상황을 공유할 뿐 아니라 끊임없는 아이디어 소통을 통해 협업 능력을 길렀습니다.

[임용진]

OV7670 카메라를 Basys3 보드에 연결하고 SCCB로 직접 레지스터를 제어하면서, 데이터시트 한 줄이 영상 품질과 동작에 얼마나 큰 영향을 주는지 체감했다.
MCU와 달리 FPGA에서는 클럭 분주, SDA/SCL 타이밍, FSM 설계를 모두 직접 맞춰야 해서 타이밍 다이어그램을 보고 설계하는 습관을 갖게 되었다. 카메라–프레임버퍼–VGA로 이어지는 경로를 직접 구성하면서, 한 픽셀, 한 클럭 단위로 시스템 전체 데이터 흐름을 바라보는 시야를 얻었다. 트러블 슈팅 부분에서도 클럭의 문제, 프로토콜의 문제가 있었지만 여러 번 고민해보고 공부 하는 과정에서 단순 구현이 아니라 레지스터로 검증 하는 공학적인 문제 해결 방식에 익숙해진 것이 가장 큰 수확이었다.

THANKS FOR PLAYING!

GAME OVER

CONTINUE? [Y / N]

[정예찬]

- 카메라 Logic 구현(Face_Write_Ctrl)
- SCCB 통신 프로토콜 구현
- VGA 주사위 구역 구현
- 보드 말 DOT 및 표정 구현
- Sound(buzzer) Effect 구현

[채민석]

- 보드게임 Core Logic 구현
- Game Effect 효과 구현
- 숫자 및 알파벳 기타 DOT 구현
- Board 및 말 모듈 구성
- 모듈 결합 구현

[이동관]

- 보드게임 Core Logic 구현
- 게임 타이틀 화면 및 플레이어 선택 화면 구현
- 말 이동 효과 추가
- 숫자 및 알파벳 도트 구현
- 게임 모듈 & 카메라 모듈 결합
- 렌더링 모듈 구현

[임용진]

- 카메라 Logic 구현
- SCCB 통신 프로토콜 구현
- 실시간 순위 배열 로직 구현
- ImageReader 모듈 구현
- OV7670 controller 구현