**Design Document: Flappy Bird Game on the DE1-SoC Board**

**Course:** CSEE W4840 - Embedded System Spring 2025
**Team Members:**

Sijun Li (sl5707), Ethan Yang(yy3526), Zidong Xu(zx2507), Tianshuo Jin(tj2591)

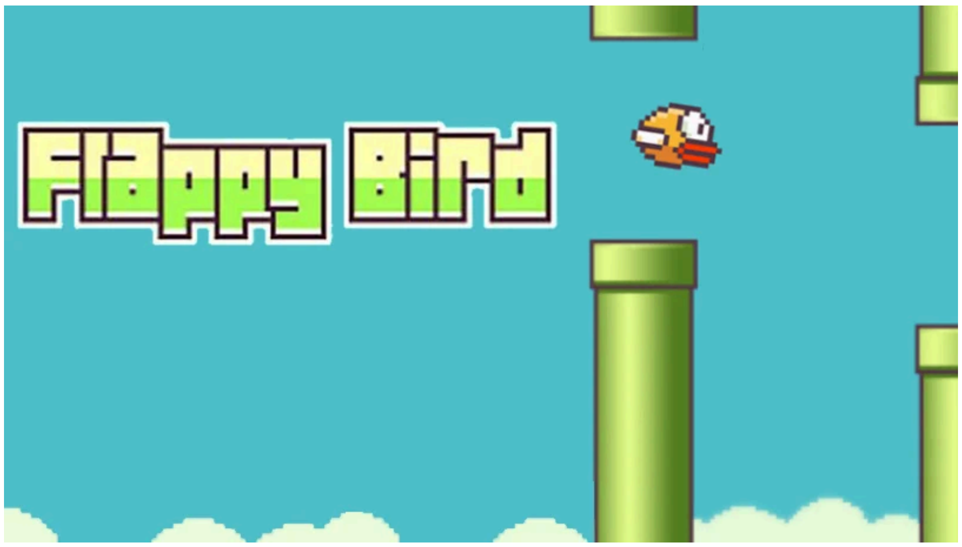
## 1. Introduction

We are implementing an enhanced version of the popular game Flappy Bird on the DE1-SoC Board. The game is a side-scroller where the player controls a bird, attempting to fly between rows of green pipes without touching them. Each safe passage through a pillar without hitting it is worth 1 point, and the game ends if it hits it. The bird briefly flaps upward each time the player presses a button; if the button is not pressed, the bird falls due to gravity.

Our implementation includes additional features to increase gameplay complexity:

- Shield power-ups are positioned in the middle of pipe gaps that provide temporary invulnerability
- Pipes that move vertically along the Y-axis as the game progresses, creating a more dynamic challenge
- Increasing speed as the player's score increases
- Infinite scrolling background simulation
- Pixel-perfect sprite-based collision detection

This project aims to implement Flappy Bird on the DE1-SoC Board, utilizing hardware-accelerated graphics rendering and software-driven game logic. To do this, it is necessary to implement graphics rendering, such as physical simulation of gravity and collision, keyboard input controlled by the player, and a scoring system.



## 2. Game Features

The FPGA-based version of Flappy Bird will include the following key features:

- **Gravity Simulation:** The bird will continuously fall due to gravity, requiring the player to press a button to jump.
- **Jump Mechanism:** A button press will apply an upward velocity to simulate a flap.
- **Dynamic Difficulty Scaling:** As the score increases, the game adjusts in three dimensions:
  - Pipe speed increases using `pipe_speed = base_speed + (score >> 3)`
  - Gap height shrinks by a few pixels per level
  - Horizontal spacing between pipes reduces to increase the reaction difficulty
- **Random Pipe Generation:** Pipes with different heights and gaps will be generated dynamically.
- **Shield Power-Up:** A Shield automatically spawns whenever the score reaches a multiple of 10. Collecting it grants 10 seconds of invincibility; if the bird collides with a pipe during this time, the shield vanishes immediately, and play continues.
- **Pixel-Perfect Collision:** Instead of basic bounding boxes, we employ bitmap masks for both the bird and obstacles. This allows for precise per-pixel collision detection, reducing false positives.
- **Moving Background:** To simulate infinite scrolling, we implemented a background layer that scrolls left using a `bg_scroll_offset`. When the offset reaches the background width, it wraps back to 0. This simulates continuous movement.
- **Pipe Vertical Oscillation:** Pipes now vertically oscillate based on a sine-wave LUT or triangle wave pattern. This motion is synchronized with the frame rate and scales with game difficulty.
- **Game States:** The game will transition between different states: Start, Playing, and Game Over.
- **Scoring System:** Start State: Awaiting player input. Play State: Active gameplay loop. Shield State: Temporary invulnerability, indicated visually. Game-Over State: Stops gameplay, displays score, and offers restart option.
- **Game Over Condition:** The game will end when a collision occurs or the bird falls out of bounds.

## 3. Software & Hardware Logic

### a. Game Logic Submodule

This is the core submodule of the game logic controller, which interfaces with all other submodules, instructing them on what to do based on the game rules. It constantly updates the screen by supplying the graphic generator with location data, score information, and game state.

i) Bird movement tracking function:

- Keeps track of the bird's position based on previous location and keyboard input
- Implements gravity physics for the bird's vertical movement
- When the control button is pressed, the bird is granted vertical upward velocity v
- The horizontal position of the bird remains constant throughout gameplay

ii) Score calculation function:

- Creates an invisible vertical line aligned with the rightmost edge of each pipe
- Score increments by 1 point when the bird completely passes the rightmost vertical edge of a pipe pair without collision
- Score display at the top of the screen updates immediately when points are earned
- Audio cue plays upon successful scoring to provide player feedback
- At certain score thresholds, like 10pts, difficulty increases are triggered

iii) Collision detection function:

- Uses bitmap-based collision detection for more precise hit detection
- Each sprite (bird, pipes, power-ups) has an associated bitmap that defines its collision area
- The bitmap contains binary values where '1' represents a solid pixel and '0' represents a transparent pixel
- When the bird's bitmap intersects with a pipe's bitmap, the game immediately ends
- If the bird's bitmap overlaps with a shield power-up's bitmap, the shield becomes active, surrounding the bird with a protective aura that prevents pipe collisions for one time
- Regardless of whether a shield is active, if the bird's bitmap makes contact with the ground, the game ends immediately

iv) Shield power-up function:

- Generates shield power-ups in the middle of pipe gaps, with new shields appearing after every 10 points scored by the player
- When the bird's bitmap overlaps with the shield power-up's bitmap, the shield activates, rendering the bird temporarily invulnerable and displaying a visual effect around the bird sprite
- The shield provides protection for exactly 10 seconds or until it absorbs one collision with a pipe, at which point the shield immediately disappears
- A frame-based counter tracks shield duration (600 frames for 10s at 60Hz) and triggers blinking via XOR modulation during the last 2–3 seconds.
- After timeout or upon absorbing a collision, the shield deactivates and updates the game state accordingly.

v) Pipe generation and movement function:

- Generates the X and Y coordinates of pipes on the screen, initially placing new pipes at the right edge of the screen with appropriately sized gaps for the bird to navigate through.

- Controls the leftward movement of pipes at a constant base speed, gradually increasing this speed as the player's score increases to enhance difficulty.
- Implements vertical movement of pipes along the Y-axis that becomes more pronounced as the game progresses, with pipes beginning to move slowly up and down once the difficulty increases.
- Places shield power-ups in the middle of pipe gaps, appearing after every 10 points scored by the player.

### b. Graphic Submodule

This submodule receives all data required for graphic generation, including:

- Coordinates of the bird, pipes, and shield power-ups
- Game state
- Score information
- Interface elements (background)

**Background Scroll Logic**: Introduce `bg_scroll_offset`, incremented every frame, wrapped around background width. Background pixel fetches use (pixel_x + bg_scroll_offset) % BG_WIDTH.

These coordinates and states are stored in memory and updated according to game logic. The memory is accessed by the graphic controller through address mapping, which then displays the necessary graphics on the screen.

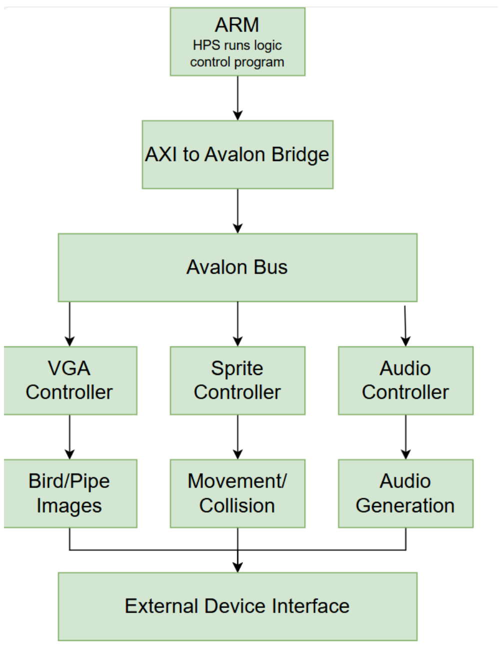
### c. Audio Submodule

The audio sounds needed in the game are encoded inside the audio generator, including:

- Collision sound when the bird hits a pipe
- Bird flapping sound
- Game over sound
- Power-up collection sound

This submodule instructs the audio controller which sound to play based on game events.

## 4. Implementation Methodology



## VGA Control

The VGA controller module is responsible for driving the VGA display output. It generates the horizontal synchronization (HSYNC) and vertical synchronization (VSYNC) signals required by the VGA timing standard, and outputs the corresponding RGB color value based on the current pixel coordinates. The module operates on a 25 MHz pixel clock and achieves a 640×480 resolution at 60 Hz refresh rate.

At each clock cycle, the module generates the current pixel coordinates `(pixel_x, pixel_y)` and determines the pixel color based on a fixed rendering priority:

1. **Bird Rendering:**
   The module first checks whether the current pixel falls within the bounding box of the bird (based on the bird's position registers and its image dimensions). If it does, the offset between the current pixel and the bird's top-left corner is used to calculate the address within the sprite ROM, from which the corresponding color is retrieved. The pixel color is only displayed if it is not the designated transparent color (e.g., RGB(255, 0, 255)), allowing the background or pipes to remain visible beneath transparent regions.

2. **Pipe Rendering:**
   If the pixel is not within the bird's area, the module checks whether it falls within the bounds of any pipe. Each pipe consists of an upper and lower section, with a vertical gap between them for the bird to pass through. Pipe positions are determined by their X coordinates and randomly generated vertical openings. If the pixel falls within a pipe region, the color is generated either from a sprite ROM or through hardcoded logic to render the green pipe graphics.

3. **Background Rendering and Scrolling:**
   If the pixel does not belong to the bird or any pipe, the background image is displayed by default. The background uses a pattern ROM to generate visuals such as sky and clouds. To simulate horizontal movement, the rendering logic includes a `bg_offset` value that adjusts the horizontal sampling position.

The pixel coordinate alignment follows a fixed-size structure. For instance, the bird image is defined as a 16×16 block, and the current bird position `(bird_x, bird_y)` is used to determine whether `(pixel_x, pixel_y)` lies within the bird's area. If so, the corresponding address in the sprite ROM is calculated, and the color is fetched. All sprite rendering strictly aligns with screen bounds to prevent memory access errors. Bird positions are maintained in integer units, with pixels as the smallest movement step.

The module enforces a strict rendering priority: bird > pipe > background. For each pixel, the controller first checks if it matches the bird sprite and is not transparent. If not, it then checks for pipe overlap, and if that also fails, it finally renders the background. This priority system is implemented with hardware combinational logic, ensuring a consistent and layered visual output even when multiple objects overlap.

The game adopts a visual design strategy where the bird remains horizontally stationary, and only its vertical position updates during jumps or falls. To simulate forward motion, the background and pipes scroll leftward at a constant speed. This scrolling effect is driven by a `scroll_offset` register, which increments over time. During rendering, `(pixel_x + scroll_offset)` is used instead of the original horizontal coordinate, creating the illusion of continuous forward movement. The X positions of the pipes are also offset accordingly, producing a dynamic scene in which obstacles approach the bird. This design eliminates the need for horizontal bird movement calculations, simplifies game logic, and improves rendering efficiency.

**Sprite Control**

The Sprite Control module manages all movable game elements, including the bird, pipes, and shields. The module updates each sprite's position and state every frame and outputs its texture index and screen coordinates to the VGA controller.

The bird maintains vertical position and velocity parameters. It continuously falls due to gravity and receives an upward initial velocity when a button press is detected. The position is updated at a fixed frame rate and moves only along the Y-axis to create a jumping effect.

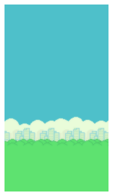
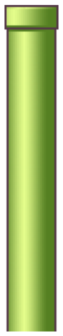
Pipes move horizontally from right to left along the X-axis. The initial movement speed is constant and increases as the player's score rises, thereby increasing the game's difficulty.

Each set of pipes consists of an upper and lower rectangular block with a gap between them for the bird to pass through. The vertical position of the gap is randomly determined at the time of pipe generation, within predefined upper and lower bounds. Once a pipe has completely exited the screen, its X coordinate is reset, and a new gap height is assigned .Pipes' vertical Y-offset is derived from a sine/triangle LUT driven by frame index and score-dependent amplitude.

Shields are generated when the player reaches specified score milestones (e.g., 20, 50, 80 points). The shield is positioned at the center of the current pipe gap and includes a lifespan control mechanism. During its lifetime, a frame counter increments each cycle, and the shield is removed either when the counter reaches a preset duration (e.g., 600 frames for 10 seconds) or upon collision. The shield's creation time is recorded, and its status is marked as active in the sprite state table.

All sprite images are stored in their respective ROMs. The Sprite Control module is responsible for managing logical states and physical coordinates but does not handle pixel-level rendering. Information such as each sprite's texture index, coordinates, and transparency flags is transmitted in real time to the VGA controller for rendering.

**Memory Budget**

| Sprite | Number of sprites | Pixel Size | Size | Example |
|---|---|---|---|---|
| Ground | 1 | 336*112 | 470B | |
| Score digits | 10 | 24*36 | 3.3KB | |
| Background(day, night) | 2 | 288*512 | 8KB |  |
| Green Pipe | 1 | 52*320 | 2.5KB |  |
| Bird | 3 | 34*24 | 1.26KB | |
| Game over | 1 | 192*42 | 758B | |

| | | | | |
|---|---|---|---|---|
| Get Ready | 1 | 184*267 | 1.6KB | FlappyBird<br>Get Ready!<br>TAP<br>TAP |

**Audio Control**

The Audio Control module generates audio output for various game events by sending signals to the onboard audio jack via PWM (Pulse Width Modulation) on the DE1-SoC Board. The module manages four distinct sound effects and background music:

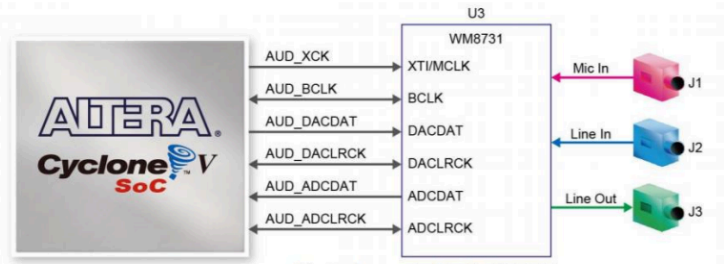
Sound Effects:

- Wing flap sound: Plays whenever the player presses the button to make the bird jump upward
- Shield collection sound: Plays when the bird collects a shield power-up
- Pipe collision sound: Plays when the bird hits a pipe (only when no shield is active)
- Death sound: Plays when the game ends (either from pipe collision without shield or hitting the ground)

The audio signals are stored in ROM as sampled waveforms at 8 kHz with 16-bit quantization for efficient memory usage while maintaining acceptable audio quality. Each sound effect has a dedicated trigger signal that is activated by the game state logic through the control register.

The module interfaces with the SSM2603 audio codec on the DE1-SoC Board through I2C for configuration and I2S for audio data transmission. Sound data is streamed from ROM to the audio codec in real-time based on game events.

Memory usage for audio is optimized by using appropriate compression techniques while maintaining sound quality that enhances the gameplay experience.



Pin Assignment of Audio CODEC

| Signal Name | FPGA Pin No. | Description | I/O Standard |
|---|---|---|---|
| AUD_ADCLRCK | PIN_K8 | Audio CODEC ADC LR Clock | 3.3V |
| AUD_ADCDAT | PIN_K7 | Audio CODEC ADC Data | 3.3V |
| AUD_DACLRCK | PIN_H8 | Audio CODEC DAC LR Clock | 3.3V |
| AUD_DACDAT | PIN_J7 | Audio CODEC DAC Data | 3.3V |
| AUD_XCK | PIN_G7 | Audio CODEC Chip Clock | 3.3V |
| AUD_BCLK | PIN_H7 | Audio CODEC Bit-stream Clock | 3.3V |
| I2C_SCLK | PIN_J12 or PIN_E23 | I2C Clock | 3.3V |
| I2C_SDAT | PIN_K12 or PIN_C24 | I2C Data | 3.3V |

## Bird Logic Module

The bird logic module is responsible for implementing the vertical movement behavior of the bird, including gravity-induced falling, jump response, boundary constraints, and state updates. This module updates the bird's position and velocity once per frame, based on the current state and user input signal, and transmits the updated coordinates to both the sprite control module and the VGA controller for rendering and collision detection.

During gameplay, the bird remains at a fixed horizontal position slightly left of the center of the screen, moving only along the vertical axis. Its position is represented by the register `bird_y` (in pixels, using integer precision), and its vertical velocity is stored in a signed register `bird_v` (in pixels per frame). The module operates on a frame-based timing schedule (e.g., 60 Hz), updating both `bird_y` and `bird_v` at the start of each frame. Gravity is applied by incrementing `bird_v` with a constant value (`gravity`) every frame, creating a continuous falling effect. The updated position is then calculated using `bird_y = bird_y + bird_v`.

When the user presses the jump button, the module immediately sets the bird's vertical velocity to a fixed negative value (e.g., -12), simulating an upward impulse. To avoid repeated triggering and signal noise, the input signal is processed with edge detection logic to respond only to rising edges. Additionally, a jump cooldown window is implemented (e.g., one jump allowed per 3 frames) to prevent multiple triggers from a long key press.

To ensure that the bird remains within the screen boundaries, the module enforces vertical position limits. If `bird_y` becomes less than 0, it is clamped to 0, and the velocity is cleared. If `bird_y` exceeds a preset maximum (e.g., screen height minus the bird's height, such as 480 - 16), the module triggers a `bird_out_of_screen` signal, which is then handled by the game state machine to initiate a failure condition. During normal gameplay, the bird's velocity may be either positive or negative and is constrained within a safe range (e.g., [-20, +20]) to prevent overflow.

If the game is in a paused or reset state (e.g., at the beginning or after a failure), the bird's position is forcibly set to an initial vertical location (e.g., vertically centered on the screen), and its velocity is reset to zero. While the game is inactive (e.g., `game_active = 0`), both position and velocity remain unchanged until the game resumes.

The entire module operates on a synchronous clock. State updates and position calculations occur at the frame rate, while jump input is processed at the system clock level to ensure responsive behavior with efficient hardware resource usage. The final outputs include the current vertical position `bird_y`, vertical velocity `bird_v`, and the out-of-bounds flag, which are used by downstream modules for collision detection and game state transitions.

**Pipe Logic Module**

The pipe logic module is responsible for managing the horizontal movement and regeneration behavior of obstacle pipes during gameplay. The module updates the position, state, and vertical gap height of each pipe on every frame, and dynamically adjusts the pipe movement speed based on the current score to gradually increase game difficulty. It supports multiple pipe pairs on screen simultaneously and ensures that their positions and spacing maintain both consistency and challenge.

Each pipe pair consists of an upper pipe and a lower pipe, with a vertical gap in between for the bird to fly through. The horizontal position of each pipe is represented by the register `pipe_x[i]`, where `i` is the index of the pipe. Pipes move from right to left at a fixed speed per frame. The initial speed is set as a constant (e.g., 2 pixels per frame), and increases with the player's score using an expression such as `pipe_speed = base_speed + (score >> 3)` to implement gradual acceleration.

When a pipe moves completely off the left edge of the screen (e.g., `pipe_x[i] + pipe_width < 0`), it is reset: its X-coordinate is reassigned to the rightmost edge of the screen, and a new random vertical gap height is generated. The gap position is stored in `pipe_gap_y[i]`, which is assigned a random value within a predefined safe range (e.g., from 100 to 380 pixels) to ensure that the bird always has a passable path. The horizontal spacing between pipe pairs is fixed to maintain rhythmic gameplay flow.

The number of active pipe pairs is set to 4 or 5, depending on the screen width and scrolling pace. Each pipe maintains information such as position, gap height, and active status, and outputs this data every frame to the VGA controller and the collision detection module. The pipe graphics are sourced from a shared sprite ROM, and rendering is handled by the VGA controller according to the real-time position and size of each pipe.

The pipe logic module does not include any vertical movement or oscillation logic. Once generated, the gap position remains fixed to simplify hardware implementation and improve control stability. The module is driven by a frame-synchronized clock and performs position updates, boundary checks, and gap regeneration at the frame rate. The final output includes each pipe's X position, gap height, and visibility status, which are used by other modules in the system.

**Collision Detection Module**

The collision detection module is responsible for determining, in real time during gameplay, whether the bird has come into contact with any pipe structure or the screen boundary, thereby identifying if the game should enter a failure state. Each frame, the module retrieves the bird's vertical position `bird_y`, and, combined with its fixed horizontal position and predefined image size, constructs the bird's rectangular boundary for the current frame. Simultaneously, it reads the horizontal positions `pipe_x[i]` and vertical gap centers `pipe_gap_y[i]` of each pipe from the pipe logic module, and calculates the upper and lower pipe regions based on the gap height and pipe dimensions.

Collision between the bird and a pipe is modeled as a rectangle overlap problem. First, a horizontal overlap check is performed to determine whether the bird's horizontal range intersects with that of the pipe. If they overlap, a vertical check follows to determine whether the bird lies outside the gap region, specifically whether `bird_y < pipe_gap_y[i] - gap_half_height` or `bird_y + bird_height > pipe_gap_y[i] + gap_half_height`. If either condition is true, a collision is considered to have occurred.

In addition to pipe collisions, the module also checks whether the bird has touched the bottom of the screen. If the bird's bottom edge (i.e., `bird_y` plus its height) exceeds the screen height, the system immediately registers a ground collision. All collision checks are implemented using combinational logic, without reliance on clock cycles or register storage, ensuring that results are computed and output in real time within each rendering frame. The output signal `collision` is asserted high whenever a collision or ground contact is detected, and is then read by the game state control module to trigger appropriate state transitions.

**Game State Control Module**

The game state control module manages transitions between different stages of gameplay, including the start, active, failure, and reset states. Based on input from the collision detection module, user interaction signals, and internal timing control, this module determines whether the game should begin, terminate, or reset. It also issues global control signals to other modules, such as position resets, score clearing, and render suspension.

Upon power-up, the system enters the "waiting to start" state, during which the bird remains at its initial position and the screen displays a static frame. When a valid jump input (e.g., a rising edge of a button signal) is detected, the state transitions to "active," enabling all motion and update logic. While the game is running, if a collision signal is received from the collision detection module, the system immediately enters the "failure" state. In this state, bird and pipe motion updates are halted, and an internal delay counter is activated to control the duration of the failure display. During this period, user inputs are ignored.

After the delay expires, the module transitions to the “reset” state, during which it issues reset signals to other modules. The bird’s position and velocity, pipe positions, score, and other system-wide variables are cleared. Once reset is complete, the system re-enters the “waiting to start” state, ready for the next game cycle. All state transitions are driven by the frame-synchronized system clock and implemented using synchronous sequential logic to ensure consistency and timing accuracy.

As the top-level controller, this module interfaces directly with the bird logic, pipe logic, collision detection, and scoring modules. Its output signals determine the overall game flow and coordinate the behavior of the entire system.

## 5. Challenges and Considerations

For future optimization, in terms of graphics, for a smooth gaming experience, the frame buffer needs to be updated efficiently to ensure a reasonable frame rate. In terms of operation, gravity and jump speed need to be balanced, and collision detection must be accurate enough to achieve a natural effect. At the same time, managing FPGA RAM frame buffer processing can minimize the delay of input response and rendering updates.

## 6. Reference

https://github.com/samuelcust/flappy-bird-assets

https://ebook.pldworld.com/_eBook/FPGA%EF%BC%8FHDL/Design%20and%20Implementation%20of%20VGA%20Controller%20Using%20FPGA/www.aicit.org/IJACT/ppl/IJACT1353PPL.pdf

https://scholars.cityu.edu.hk/files/26815494/FPga_Based_high_Performance_collision_Detection_an_enabling_Technique_for_image_guided_robotic_surgery.pdf