SUNY at Albany

Department of Electrical and Computer Engineering

November 2025

ECE 231: Digital Systems

RTL Digital Design Project: Flappy Bird Processor

High-Level State Machine Implementation

Author(s):
Saugat Shah
Gwanghoon Lee

Instructor:
Professor Dave Ardrey

Contents

1 Project Summary

This section details the hardware design and simulation of a Flappy Bird game on a custom processor. The system architecture utilizes a dedicated controller, implemented as a moore-finite-state machine (FSM), paired with a structured datapath to execute the core game logic. The complete design was captured in VHDL and its functional correctness was validated through simulation using the Vivado toolkit.

1.1 Project Overview

In this project, the objective was to design a custom processor to emulate a functional Flappy Bird-like game. Specifically, the focus was on implementing the following capabilities:

- Make a character have a positive upward velocity (jump) when a button (B) is pushed
- Implementing basic free fall (acceleration downward)
- Having character die when they hit the ground

1.2 Design Approach

The implementation followed a structured digital design process:

- High-Level State Machine (HLSM) specification
- Datapath design with appropriate registers and logical units
- Translation of the HLSM into a Moore Machine design
- Implementation of the Moore Machine using a custom controller architecture
- Integration of datapath and controller into complete processor using control lines
- VHDL implementation and simulation verification for the custom processor

1.3 Implementation Results

The final design successfully implemented all required functionalities including jump initiation on button press, accelerating downward motion, and game-over detection when the character hits the ground level. The processor utilized the following components for implementation:

Component	Purpose/Function
6-bit Components	
6-bit Equality Comparator	Compares 6-bit values
YVel Register	Stores vertical velocity
YPos Register	Stores vertical position
6-bit Ripple Carry Adder	6-bit addition
6-bit Ripple Carry Subtractor	6-bit subtraction
Position Multiplexer (pos_mux)	Position data selection
Velocity Multiplexer (vel_mux)	Velocity data selection
Single-bit Components	
3-input AND Gate	3-input AND operation
2-input AND Gate	2-input AND operation
3-input OR Gate	3-input OR operation
2-input OR Gate	2-input OR operation
NOT Gate	Logical inversion

Table 1: Processor Components by Bit

2 Functional Description

This section details the functional requirements and behavioral specifications of the Flappy Bird custom processor, including input/output definitions and the high-level state machine description.

2.1 System Overview

The purpose of this custom processor is to serve as a dedicated hardware core that executes the fundamental logic of a Flappy Bird-style game. It continuously calculates the game character's flight velocity and Y-position, detects collisions with the ground, and manages game state transitions. This is all implemented through digital logic using a datapath-controller architecture based on the HLSM shown in Figure 1.

2.2 Input/Output Specification

Inputs:

- `clk` - System clock signal [1-Bit]
- `B` - Jump button input [1-Bit]
- `ground_level` - input specifying current ground height [6-Bit]

Outputs:

- `J` - Output indicating jump state [1-Bit]
- `F` - Output indicating free fall state [1-Bit]
- `D` - Output indicating dead/game over state [1-Bit]
- `YPos` - Output showing character's current vertical position [6-Bit]

Local Storage:

- `YPosReg` - Register storing current Y position [6-Bit]
- `YVelReg` - Register storing current Y velocity [6-Bit]

2.3 Input/Output Relationship

The processor maintains the following input-output relationships:

- `B` (button press) directly controls jump initiation when in Die or Fall states
- `ground_level` determines collision detection threshold for Die state transition
- `YPos` output reflects real-time vertical position calculated from velocity integration
- State outputs (J, F, D) are mutually exclusive and indicate the current operational mode

The controller ensures that outputs maintain correct timing relationships with inputs, with all state transitions synchronized to the clock edge.

2.4 High-Level State Machine

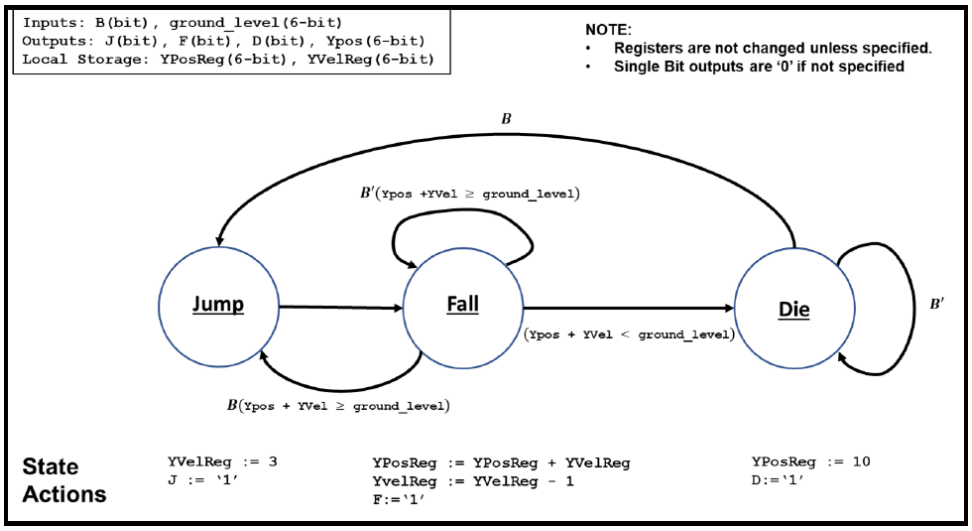

Figure 1: High-Level State Machine Diagram for Flappy Bird Processor

The HLSM consists of **3 main states:**

1. **Jump State**
   - Character Y-Position increases upward by 3 units
   - Triggered by button press (B)
   - Y-Velocity register updated to '3'
   - J output signal = 1
2. **Fall State**
   - Character accelerates downward with velocity -1 for each cycle
   - Y-Position and Y-Velocity registers update continuously
   - Y-Position is calculated by adding the Y-Velocity and characters Y-Position
   - F output signal = 1
3. **Die State**
   - Game over when character hits ground
   - User can press B to restart
   - Character's Y-Position is reset to '10'
   - D output signal = 1

3 Circuit Design

This section documents the complete circuit design process, including datapath implementation, controller development, and final processor integration.

Note: All components may be referenced to Table 1.

3.1 Datapath Design

3.1.1 Design Process

The datapath was systematically designed by analyzing the HLSM operations:

- *Operation Identification*: Extracted all mathematical operations (addition, subtraction, comparison)
- *Register Requirements*: Identified needed storage (YPos and YVel Registers)
- *Data Flow Mapping*: Created connections between arithmetic units and registers
- *Conditional Logic*: Implemented multiplexers for HLSM conditional operations
- *Control Signal Integration*: Added control line interfaces for controller communication

3.1.2 Implementation Details

The datapath implements all multi-bit processes outlined in the HLSM (Figure 1) including:

- 6-bit registers for Y-Position and Y-Velocity storage
- Adder to calculate new Y-Position (YPos + YVel)
- Subtractor using 2's complement addition for Y-Velocity updates
- Equality comparator for ground collision detection
- Multiplexers for conditional operations (position reset and velocity jump)

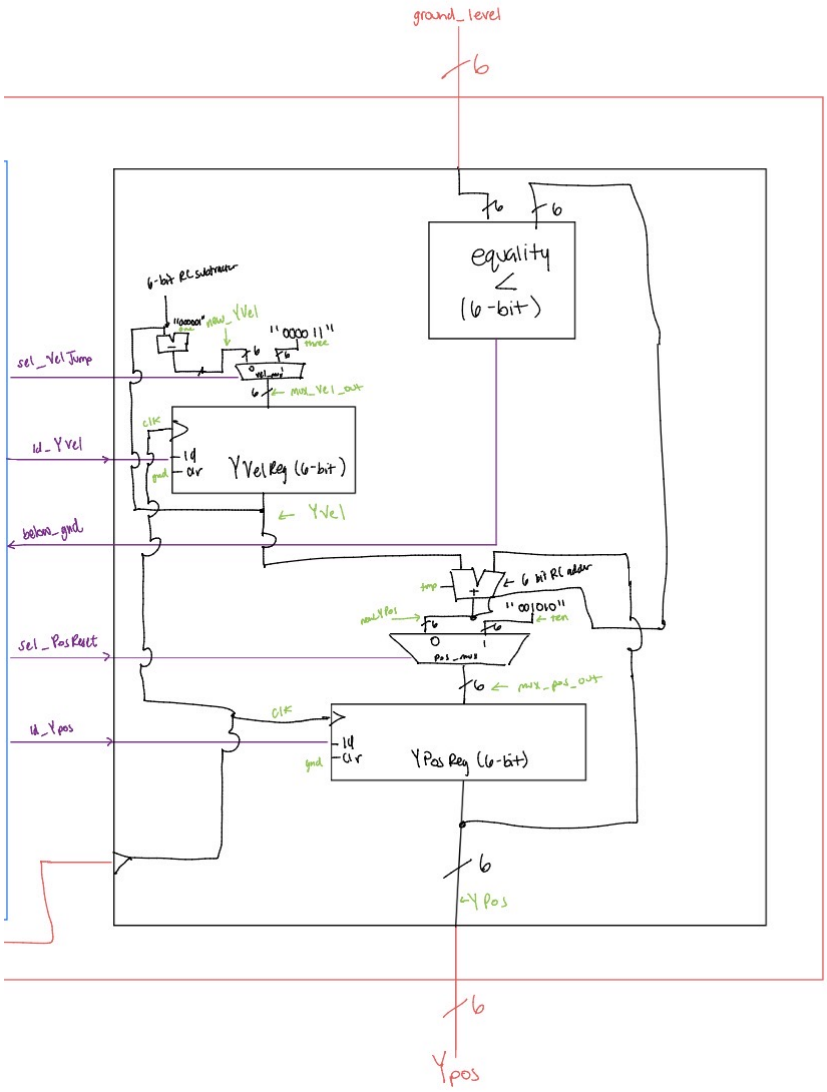

Figure 2: Datapath Schematic Diagram

The datapath uses control lines (purple) to compute and influence processor decisions.

3.2 Controller Design

3.2.1 HLSM to FSM Translation

The controller was derived from the HLSM through systematic conversion:

- *State Mapping*: Each HLSM state (Jump, Fall, Die) became an FSM state
- *Condition Translation*: HLSM conditions (B, below gnd) became FSM transition conditions
- *Output Mapping*: HLSM operations translated to control output signals for datapath

3.2.2 FSM Implementation

The controller implements the finite state machine derived from the HLSM, generating appropriate control signals for the datapath. The purple control lines in Figure 2 correspond directly to operations specified in the HLSM:

- `sel_PosReset` controls HLSM operation "YPos ← 10"
- `sel_VelJump` controls HLSM operation "YVel ← 3"
- `ld_YPos` and `ld_YVel` enable register updates per HLSM timing

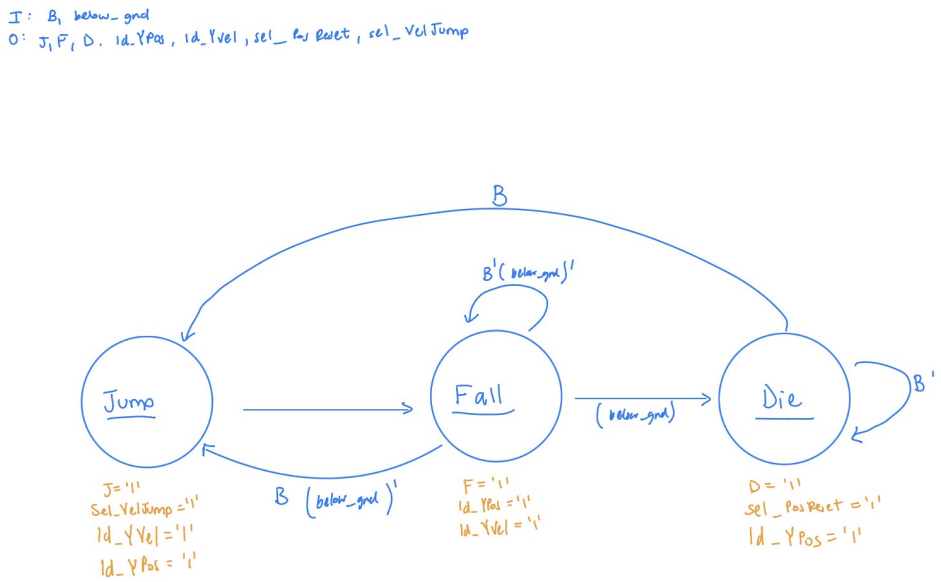

Figure 3: Finite State Machine Controller Diagram

3.3 Processor Integration

3.3.1 Custom Processor Architecture Implementation

The complete processor follows the custom processor architecture pattern:

- **Datapath Component**: Contains all data processing and storage elements
- **Controller Component**: Implements control logic and state management
- **Control Lines**: Communication interface between controller and datapath

3.3.2 Bottom-Up Implementation Strategy

The processor was implemented using a bottom-up approach:

Datapath → Next State Logic → Output Logic → Controller → Processor

Implementation Steps:

1. Datapath module implemented with all arithmetic and storage components
2. Next State Logic designed using minimized Boolean expressions from state table
3. Output Logic implemented to generate correct state indicators (J, F, D)

4. Controller assembled by integrating next state and output logic modules
5. Complete processor created by connecting datapath and controller

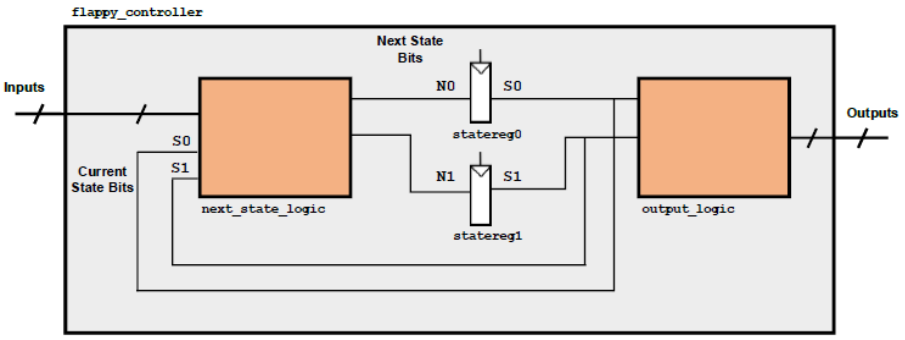

Figure 4: Complete Controller Template

The following encoding for the structure is described below:

State	S_1	S_0
Die	0	0
Jump	0	1
(XXXX)	1	0
Fall	1	1

Table 2: State Encoding for FSM Controller

The controller was then implemented in VHDL using the minimal boolean expressions as seen in Figures 5 and 6:

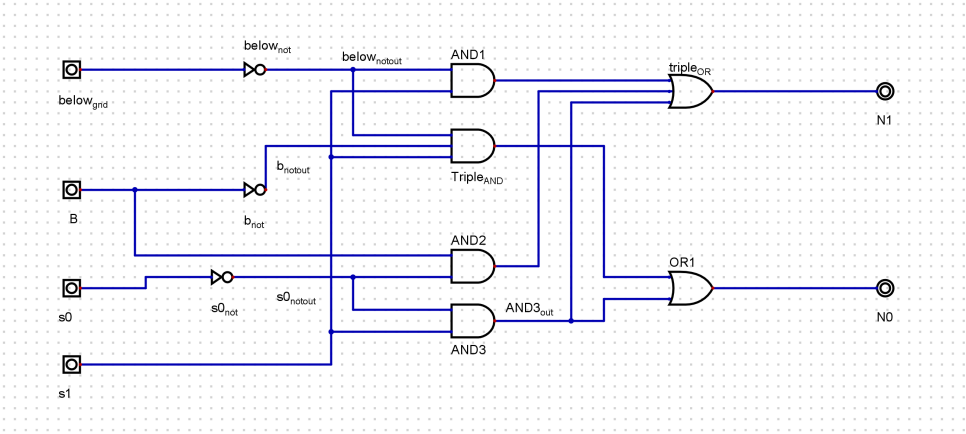

Figure 5: Next State Logic Schematic

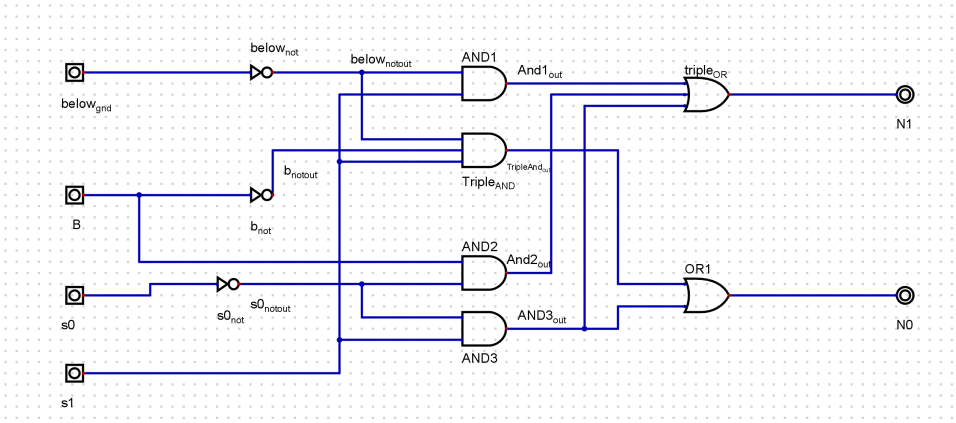

Figure 6: Output Logic Schematic

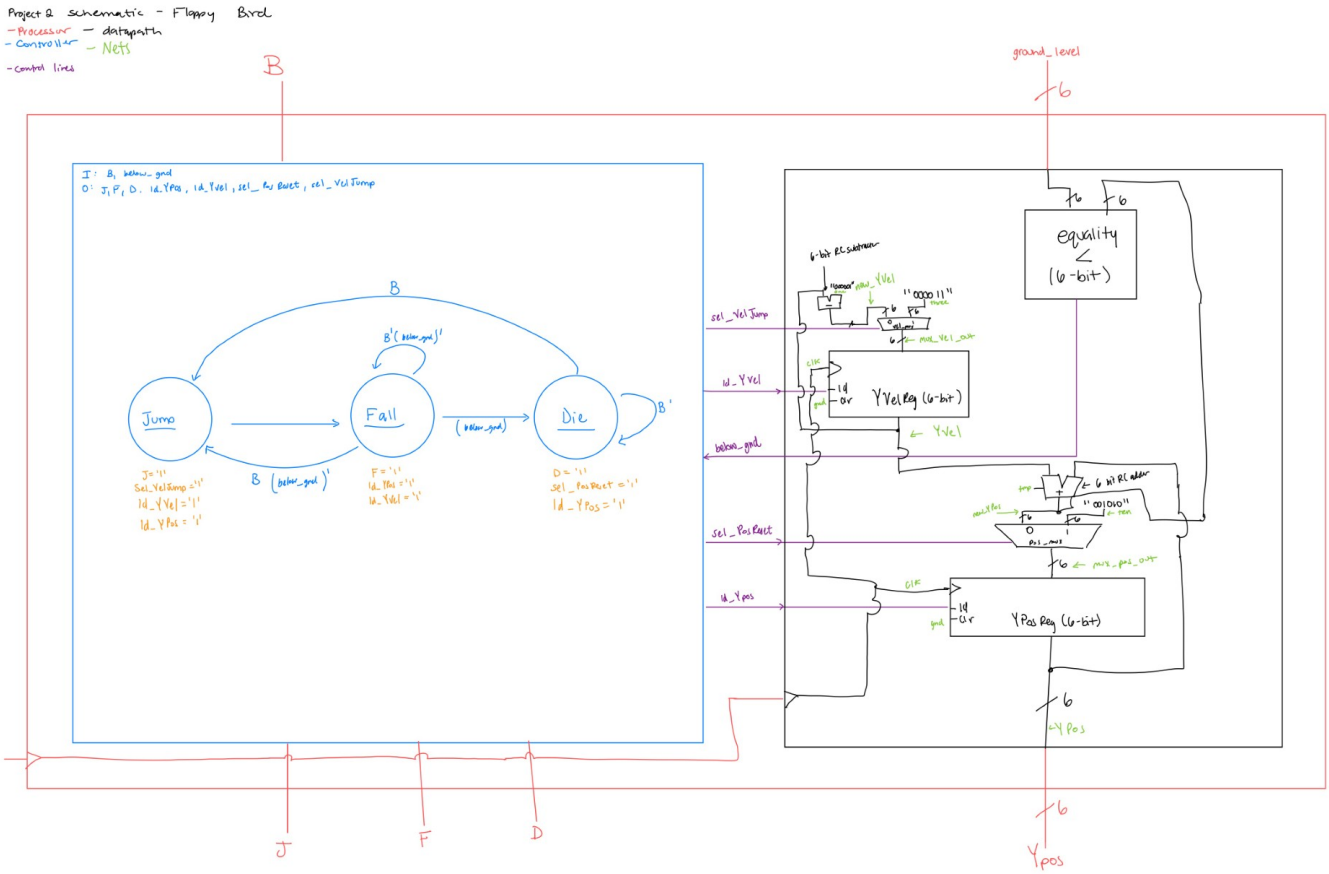

Figure 7: Complete Processor Schematic With Abstracted Controller

4 Simulation Results

This section presents simulation results for the datapath, controller, and the complete processor and explains validation for each section.

4.1 Datapath Simulation

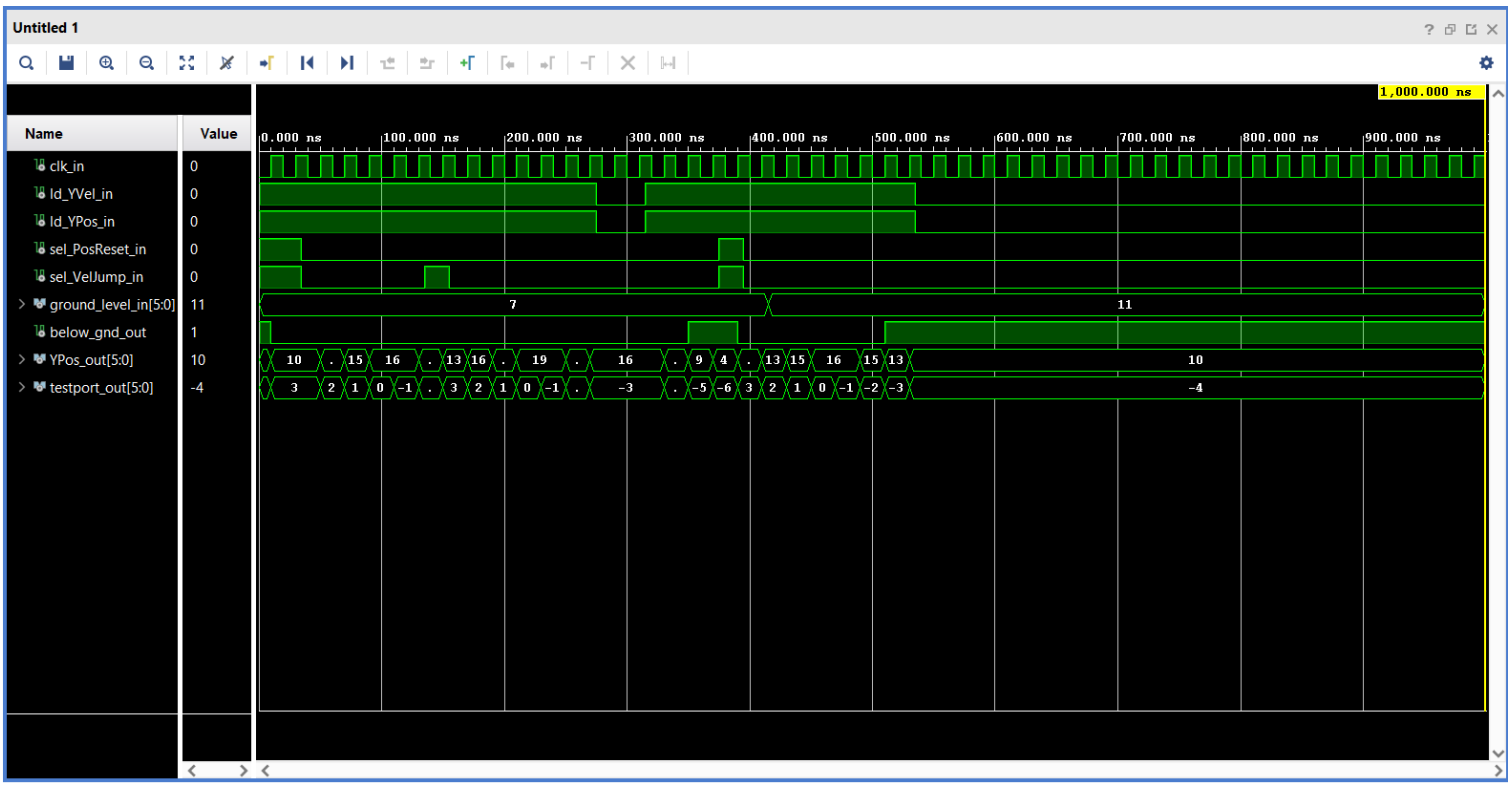

Figure 8: Datapath Testbench Simulation Results

Key Observations:

- Position reset to 10 occurs when `sel_PosReset` is HIGH (∼0 ns)
- Velocity jump to 3 occurs when `sel_VelJump` is HIGH (∼50 ns)
- The registers are NOT updated when `ld_YPos` or `ld_YVel` is LOW (∼270 ns)
- `below_gnd` detection is HIGH when the character's `YPos` < `ground_level_in` (∼510 ns)

This successfully verifies that the datapath works in accordance to the HLSM

4.2 Controller Simulation

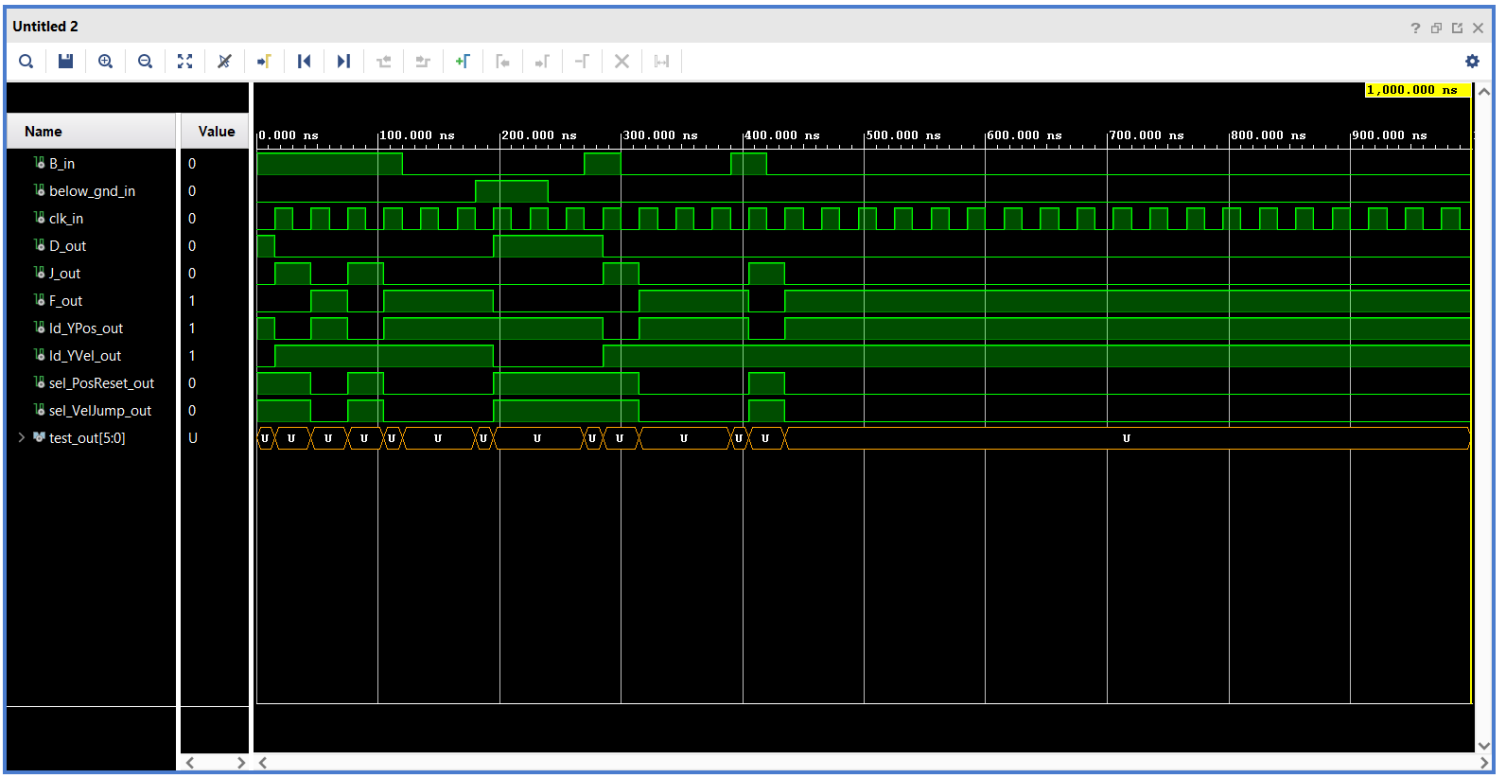

Figure 9: Controller Testbench Simulation Results

Key Observations:

- System enters state J when B is pressed (∼20 ns)
- System enters state F when B is not pressed (∼120 ns)
- System enters state D when the system obtains signal `below_gnd` (∼190 ns)
- System accurately gives output signals according to the FSM in Figure 3 `ld_YPos` or `ld_YVel` (∼190 ns)
- System accurately gives output multiplexer according to the FSM in Figure 3 `sel_PosReset` or `sel_VelJump` (∼70 ns)

This successfully verifies that the controller works in accordance to the FSM

4.3 Processor Simulation

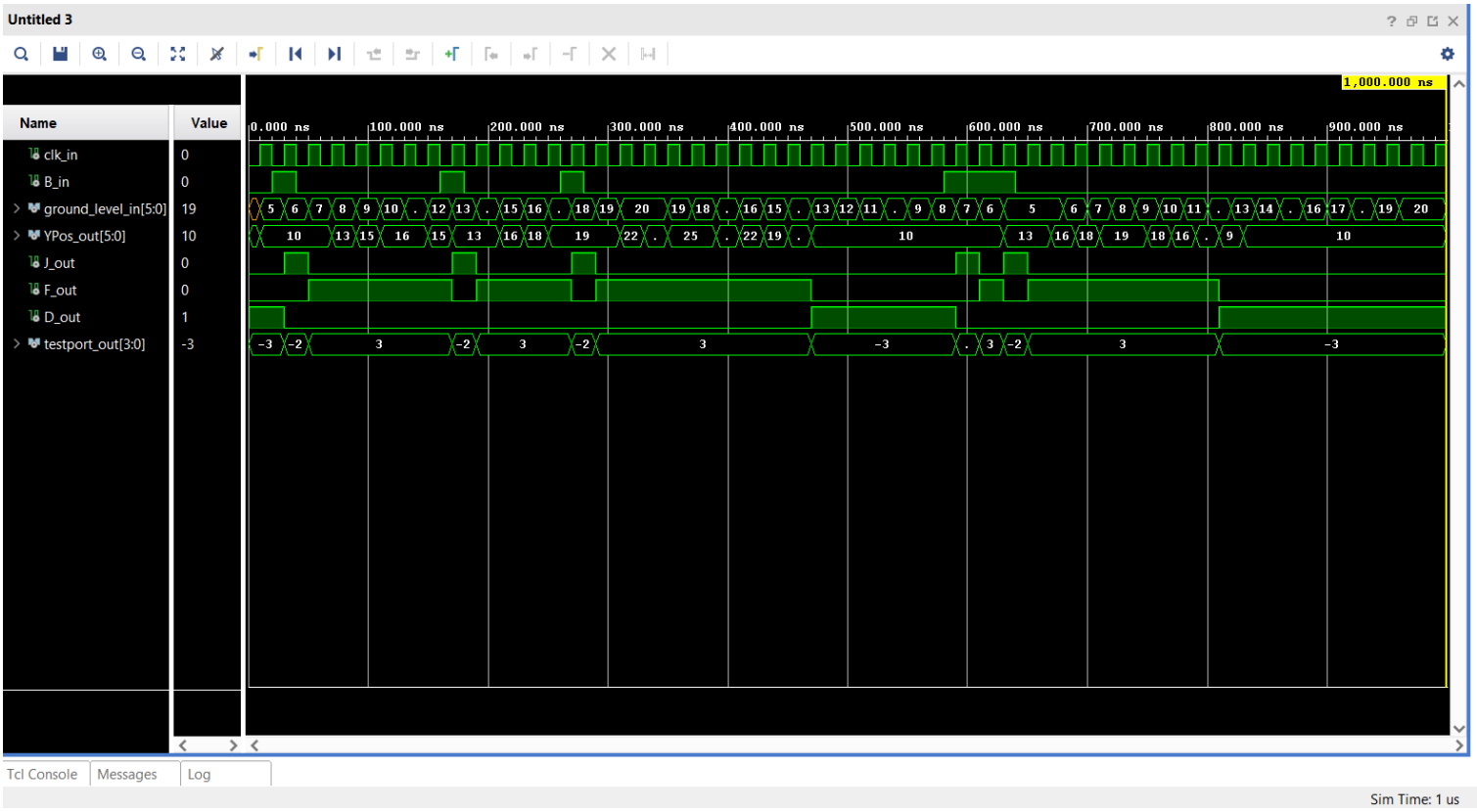

Figure 10: Complete Processor Testbench Simulation Results

Key Observations:

- Character responds to button presses with upward jumps '+3' (∼30 ns)
- Free fall acceleration by '-1' is properly implemented (∼50 ns to ∼170 ns)
- Game over detection triggers when character hits ground and game does not restart until user presses B again (∼470 ns to ∼590 ns)
- All states transition according to specification in the HLSM

This successfully verifies that the controller and datapath work together to create a fully functioning custom processor in accordance with the HLSM

5 Conclusion

This section summarizes the project outcomes and full design process.

5.1 Project Summary

The project involved designing a custom processor to emulate a Flappy Bird-style game using digital logic principles. The core functionality included implementing character jumps on button press, increasing downward acceleration, and position detection with the ground. The design followed a structured hardware development approach using a datapath-controller architecture, with all components implemented in VHDL and validated through simulation.

5.2 Design Approach

We applied a systematic digital design methodology that started with a High-Level State Machine (HLSM) specification defining the three main states: Jump, Fall, and Die. This HLSM was translated into a Moore finite state machine for the controller. The datapath was designed with multi-bit components for position/velocity calculations and control lines to communicate with the controller. The controller was then implemented using Next State and Output Logic expressions. Finally, a bottom-up implementation strategy was used, building from basic components to the complete processor, with thorough simulation testing at each stage for complete verification.

5.3 Results and Implementation

- Character jumps (+3 units) triggered by button presses
- Downward acceleration (-1 per clock cycle)
- Accurate ground collision detection
- Proper state transitions between Jump, Fall, and Die modes in accordance with the HLSM

All simulations confirmed valid functionality, with the datapath performing arithmetic operations accurately, the controller generating appropriate control signals, and the integrated processor executing the complete game logic according to specifications in the HLSM.

5.4 Future Improvements

- Create testports for pinpoint testing of various logical components
- Use don't-care states accurately to further simplify boolean expressions
- Implement more comprehensive error checking and validation procedures
- Add additional debugging features to streamline future development

6 Appendix: VHDL Code

6.1 VHDL Code by Task

- **Task 2 (Datapath):** flappy_datapath.vhd
- **Task 3 (Controller):** flappy_controller.vhd
- **Task 4 (Processor):** flappy_processor.vhd

6.2 Datapath VHDL Code (Task 2)

Figure A1: flappy_datapath.vhd Implementation

```
-- Component to implement all necessary mathematical and multi-bit operations
-- for the flappy game simulator project.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity flappy_datapath is
  Port (clk:  in std_logic;
        ld_YPos: in std_logic; -- Updates YPosReg when 1
        ld_YVel: in std_logic; -- Updates YVelReg when 1
        sel_PosReset: in std_logic; -- Select bit: 1 means reset YPos to 10
        sel_VelJump: in std_logic; -- Select bit: 1 means set YVel to 3
        below_gnd: out std_logic; -- Character is below ground (YPos < ground_level)
        ground_level: in std_logic_vector (5 downto 0);
        YPos: inout std_logic_vector (5 downto 0);
        testport: out std_logic_vector (5 downto 0));
end flappy_datapath;

architecture structural of flappy_datapath is

-- Components we need (add below):
component lessthan_6bit
Port ( A : in STD_LOGIC_VECTOR (5 downto 0);
           B : in STD_LOGIC_VECTOR (5 downto 0);
           E : out STD_LOGIC);
end component;

component six_bit_load_register
    Port ( ld : in STD_LOGIC;
           clr : in STD_LOGIC;
           clk : in STD_LOGIC;
           D : in STD_LOGIC_VECTOR(5 downto 0);  -- 6-bit data input
           Q : out STD_LOGIC_VECTOR(5 downto 0)); -- 6-bit data output
end component;

component six_bit_ripple_carry_adder
    Port ( X : in STD_LOGIC_VECTOR(5 downto 0);
           Y : in STD_LOGIC_VECTOR(5 downto 0);
           S : out STD_LOGIC_VECTOR(5 downto 0);
           C_Out : out STD_LOGIC);
end component;

component six_bit_two_to_one_multiplexer
```

```
    Port ( A_in : in STD_LOGIC_VECTOR(5 downto 0);
           B_in : in STD_LOGIC_VECTOR(5 downto 0);
           S_in : in STD_LOGIC;
           Q_out : out STD_LOGIC_VECTOR(5 downto 0));
end component;

component six_bit_ripple_carry_subtractor
    Port ( X : in STD_LOGIC_VECTOR(5 downto 0);
           Y : in STD_LOGIC_VECTOR(5 downto 0);
           S : out STD_LOGIC_VECTOR(5 downto 0);
           C_Out : out STD_LOGIC);
end component;

-- hardcoded signals we need (add others as needed):
signal gnd: std_logic := '0'; -- to hardcode inputs to 0
signal tmp: std_logic; -- to be tied to unused outputs.
signal three: std_logic_vector(5 downto 0) := "000011"; -- hardcoded
   number three
signal one: std_logic_vector(5 downto 0) := "000001"; -- hardcoded
   number one
signal ten: std_logic_vector(5 downto 0) := "001010"; -- hardcoded
   number 10

-- Additional signals for internal nets
signal YVel: std_logic_vector (5 downto 0); -- signal to use as output
   of velocity register
-- Add other signals for interal nets as needed
signal new_YPos: std_logic_vector (5 downto 0); -- new position
   calculation
signal new_YVel: std_logic_vector (5 downto 0); -- new velocity
   calculation
signal mux_vel_out: std_logic_vector (5 downto 0); -- velocity mux
   output
signal mux_pos_out: std_logic_vector (5 downto 0); -- position mux
   output


begin

-- Add instances of components here
vel_reg: six_bit_load_register port map (ld => ld_YVel, clr => gnd, clk
    => clk, D => mux_vel_out, Q => YVel);
pos_reg: six_bit_load_register port map (ld => ld_YPos, clr => gnd, clk
    => clk, D => mux_pos_out, Q => YPos);
equality: lessthan_6bit port map (A => new_YPos, B => ground_level, E
   => below_gnd);
adder: six_bit_ripple_carry_adder port map (X => YVel, Y => YPos, S =>
   new_YPos, C_out => tmp);
subtractor: six_bit_ripple_carry_subtractor port map ( X => YVel, Y =>
   one, S => new_YVel, C_out => tmp);
vel_mux: six_bit_two_to_one_multiplexer port map (A_in => new_YVel,
   B_in => three, S_in => sel_VelJump, Q_out => mux_vel_out);
pos_mux: six_bit_two_to_one_multiplexer port map (A_in => new_YPos,
   B_in => ten, S_in => sel_PosReset, Q_out => mux_pos_out);

-- The testport can be used for evaluating internal nets.
--    Test a bus
testport <= YVel; -- default . . . output velocity value.
```

```
86
87 --    Test individual bits
88 --    testport(0) <= gnd;
89
90 end structural;
```

6.3 Controller VHDL Code (Task 3)

Figure A2: flappy_controller.vhd Implementation (Next State, Output Logic)

```
1  --START OF NEXT STATE LOGIC CODE
2
3
4  -- This is a combinational logic circuit mapping all inputs and current
       state
5  -- to the next state.
6
7  library IEEE;
8  use IEEE.STD_LOGIC_1164.ALL;
9
10 entity flappy_nextstate_logic is
11     Port ( B : in STD_LOGIC;
12            below_gnd : in STD_LOGIC;
13            S0 : in STD_LOGIC;
14            S1 : in STD_LOGIC;
15            N0 : out STD_LOGIC;
16            N1 : out STD_LOGIC;
17            testport : out std_logic);
18 end flappy_nextstate_logic;
19
20 architecture Structural of flappy_nextstate_logic is
21
22 -- Componenents needed:
23 component not_gate
24     Port ( A : in STD_LOGIC;
25            Q : out STD_LOGIC);
26 end component;
27
28 component two_input_and_gate
29     Port ( A : in STD_LOGIC;
30            B : in STD_LOGIC;
31            Q : out STD_LOGIC);
32 end component;
33
34 component two_input_or_gate
35     Port ( A : in STD_LOGIC;
36            B : in STD_LOGIC;
37            Q : out STD_LOGIC);
38 end component;
39
40 component three_input_and_gate
41     Port ( A : in STD_LOGIC;
42            B : in STD_LOGIC;
43            C : in STD_LOGIC;
44            Q : out STD_LOGIC);
```

```
45 end component;
46
47 component three_input_or_gate
48     Port ( A : in STD_LOGIC;
49            B : in STD_LOGIC;
50            C : in STD_LOGIC;
51            Q : out STD_LOGIC);
52 end component;
53
54
55 -- Hard coded signals
56 signal vdd : std_logic := '1';
57 signal gnd : std_logic := '0';
58
59 -- Internal signals;
60 signal below_not_out: std_logic;
61 signal B_not_out: std_logic;
62 signal S0_not_out: std_logic;
63 signal AND1_out: std_logic;
64 signal AND2_out: std_logic;
65 signal AND3_out: std_logic;
66 signal triple_and_out: std_logic;
67
68
69
70 begin
71
72 -- Instances of components
73 below_not: not_gate port map (below_gnd, below_not_out);
74 b_not: not_gate port map (B, B_not_out);
75 S0_not: not_gate port map (S0, S0_not_out);
76 AND1: two_input_and_gate port map (below_not_out, S1, AND1_out);
77 AND2: two_input_and_gate port map (B, S0_not_out, AND2_out);
78 AND3: two_input_and_gate port map (S0_not_out, S1, AND3_out);
79 triple_and: three_input_and_gate port map (below_not_out, b_not_out, S1
       , triple_and_out);
80 OR1: two_input_or_gate port map (triple_and_out, AND3_out, N0);
81 triple_OR: three_input_or_gate port map (AND1_out, AND2_out, AND3_out,
       N1);
82
83
84 -- Single bit test port
85 testport <= gnd;
86
87 end Structural;
88
89 --START OF OUTPUT LOGIC CODE
90
91 -- This is a combinational logic circuit mapping current state to
92 -- to all outputs of the controller
93
94 library IEEE;
95 use IEEE.STD_LOGIC_1164.ALL;
96
97 entity flappy_output_logic is
98     Port ( S0: in std_logic;
99            S1: in std_logic;
100           J : out STD_LOGIC;
```

```
            F : out STD_LOGIC;
            D : out STD_LOGIC;
            ld_YPos : out STD_LOGIC;
            ld_YVel : out STD_LOGIC;
            sel_PosReset: out std_logic;
            sel_VelJump: out std_logic;
            testport : out std_logic);
end flappy_output_logic;


architecture Structural of flappy_output_logic is

-- Componenents needed:
component not_gate
    Port ( A : in STD_LOGIC;
           Q : out STD_LOGIC);
end component;

component two_input_and_gate
    Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           Q : out STD_LOGIC);
end component;

component two_input_or_gate
    Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           Q : out STD_LOGIC);
end component;


-- Hard coded signals
signal vdd : std_logic := '1';
signal gnd : std_logic := '0';

-- Internal signals;
signal S0_not_out : std_logic;
signal S1_not_out : std_logic;
signal jump_out : std_logic;
signal die_out : std_logic;


begin

-- Instances of components / output assignments
S0_not: not_gate port map (S0, S0_not_out);
S1_not: not_gate port map (S1, S1_not_out);
And1: two_input_and_gate port map (S0_not_out, S1, jump_out);
And2: two_input_and_gate port map (S0_not_out, S1_not_out, die_out);
And3: two_input_and_gate port map (S0, S1, F);
OR1: two_input_or_gate port map (S0, S1_not_out, ld_YPos);

sel_VelJump <= S0_not_out;
sel_PosReset <= S0_not_out;
J <= jump_out;
D <= die_out;
ld_YVel <= S1;

```

```
159
160 -- Single bit test port
161 testport <= gnd;
162
163 end Structural;
```

6.4 Processor VHDL Code (Task 4)

Figure A3: flappy_processor.vhd Implementation

```
1  -- Highest level of flappy game simulator.  Connections between
       controller and datapath
2
3  library IEEE;
4  use IEEE.STD_LOGIC_1164.ALL;
5
6  entity flappy_processor is
7      Port ( CLK : in STD_LOGIC ;
8             B : in STD_LOGIC;
9             ground_level : in STD_LOGIC_VECTOR (5 downto 0);
10            J : out STD_LOGIC;
11            F : out STD_LOGIC;
12            D : out STD_LOGIC;
13            YPos : inout STD_LOGIC_VECTOR (5 downto 0);
14            testport : out std_logic_vector (3 downto 0));
15 end flappy_processor;
16
17 architecture Behavioral of flappy_processor is
18
19 component flappy_controller is
20     Port ( CLK : in STD_LOGIC;
21            B : in STD_LOGIC;
22            below_gnd : in STD_LOGIC;
23            J : out STD_LOGIC;
24            F : out STD_LOGIC;
25            D : out STD_LOGIC;
26            ld_YPos : out STD_LOGIC;
27            ld_YVel : out STD_LOGIC;
28            sel_PosReset: out std_logic;
29            sel_VelJump: out std_logic;
30            testport : out std_logic_vector (5 downto 0));
31 end component;
32
33 component flappy_datapath is
34   Port (clk:  in std_logic;
35         ld_YPos: in std_logic;
36         ld_YVel: in std_logic;
37         sel_PosReset: in std_logic;
38         sel_VelJump: in std_logic;
39         below_gnd: out std_logic;
40         ground_level: in std_logic_vector (5 downto 0);
41         YPos: inout std_logic_vector (5 downto 0);
42         testport: out std_logic_vector (5 downto 0));
43 end component;
44
```

```
45 -- IO signals
46 signal ld_YPos_net, ld_YVel_net, sel_PosReset_net, sel_VelJump_net,
       below_gnd_net  : std_logic;
47 signal control_test, data_test : std_logic_vector (5 downto 0);
48
49 begin
50
51 controller: flappy_controller port map (CLK, B, below_gnd_net, J, F, D,
       ld_YPos_net, ld_YVel_net,
52                                          sel_PosReset_net,
       sel_VelJump_net, control_test);
53 datapath: flappy_datapath port map (CLK, ld_YPos_net, ld_YVel_net,
       sel_PosReset_net, sel_VelJump_net, below_gnd_net,
54                                      ground_level,  YPos, data_test);
55
56 -- Debug port (can be modified by students)
57 testport(0) <= ld_YPos_net;
58 testport(1) <= ld_YVel_net;
59 testport(2) <= sel_PosReset_net;
60 testport(3) <= sel_VelJump_net;
61
62 end Behavioral;
```

7 Appendix: AI Usage Documentation

This appendix documents any generative AI tools used in accordance with course policies.

AI Usage Statement: No generative AI tools were used in the design, implementation, or documentation of this project. All work was completed independently in accordance with course policies.