

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

Instructor:

Professor Dave Ardrey



**UNIVERSITY
AT ALBANY**

STATE UNIVERSITY OF NEW YORK

Contents

1 Project Summary	3
Part 1: Project Summary	3
1.1 Project Overview	3
1.2 Design Approach	3
1.3 Implementation Results	3
2 Functional Description	5
Part 2: Functional Description	5
2.1 System Overview	5
2.2 Input/Output Specification	5
2.3 Input/Output Relationship	5
2.4 High-Level State Machine	6
3 Circuit Design	7
Part 3: Circuit Design	7
3.1 Datapath Design	7
3.1.1 Design Process	7
3.1.2 Implementation Details	7
3.2 Controller Design	8
3.2.1 HLSM to FSM Translation	8
3.2.2 FSM Implementation	9
3.3 Processor Integration	9
3.3.1 Custom Processor Architecture Implementation	9
3.3.2 Bottom-Up Implementation Strategy	9
4 Simulation Results	12
Part 4: Simulation Results	12
4.1 Datapath Simulation	12
4.2 Controller Simulation	13
4.3 Processor Simulation	14
5 Conclusion	15
Part 5: Conclusion	15
5.1 Project Summary	15
5.2 Design Approach	15

5.3	Results and Implementation	15
5.4	Future Improvements	15
6	Appendix: VHDL Code	16
	Appendix: VHDL Code	16
6.1	VHDL Code by Task	16
6.2	Datapath VHDL Code (Task 2)	16
6.3	Controller VHDL Code (Task 3)	18
6.4	Processor VHDL Code (Task 4)	19
7	Appendix: AI Usage Documentation	21
	Appendix: AI Usage Documentation	21

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

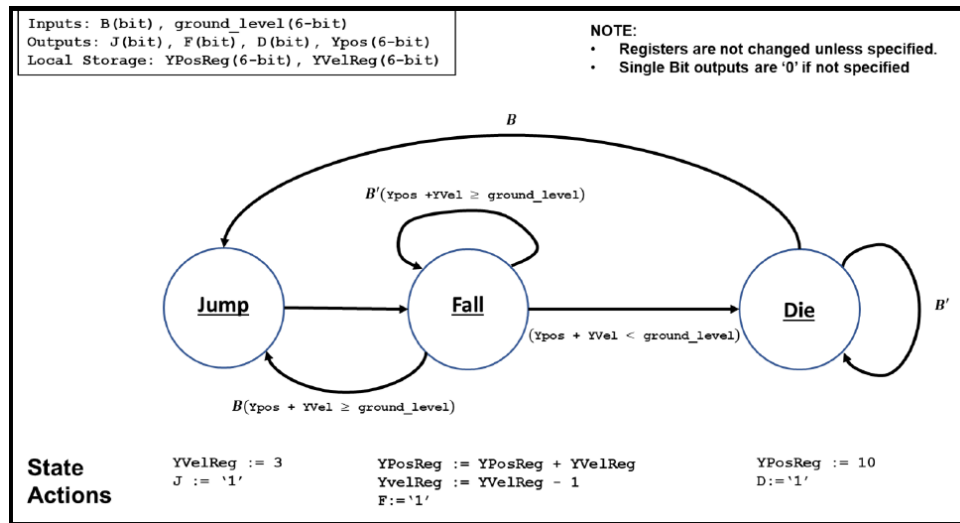


Figure 1: High-Level State Machine Diagram for Flappy Bird Processor

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)

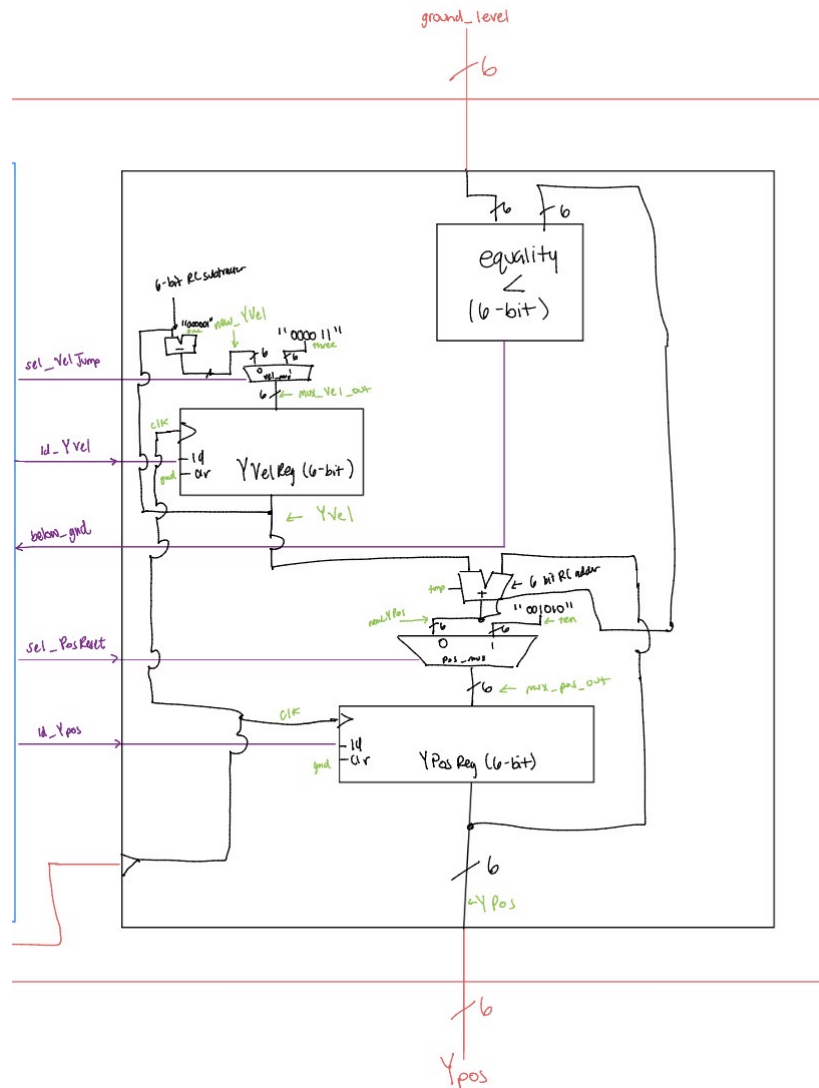


Figure 2: Datapath Schematic Diagram

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 " $YPos \leftarrow 10$ "
- `sel_VelJump` controls HLSM operation " $YVel \leftarrow 3$ "
- `ld_YPos` and `ld_YVel` enable register updates per HLSM timing

$I: B, \text{below_grd}$
 $O: J, F, D, \text{ld_YPos}, \text{ld_YVel}, \text{sel_PosReset}, \text{sel_VelJump}$

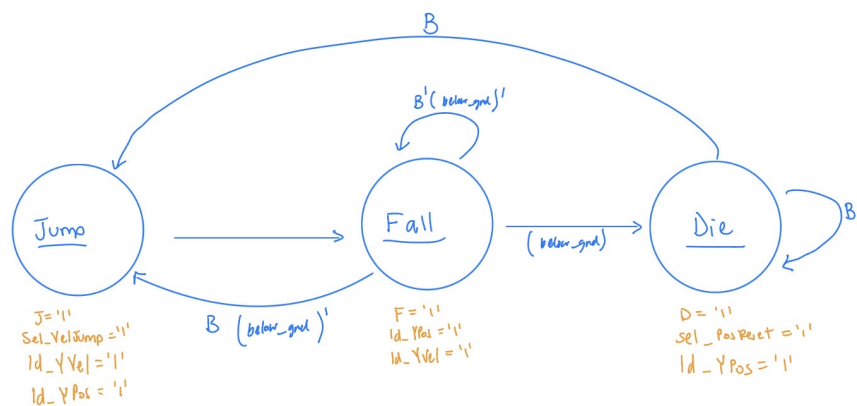


Figure 3: Finite State Machine Controller Diagram

3.3 Processor Integration

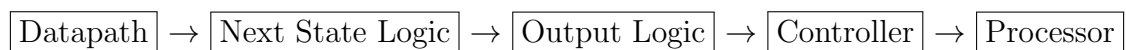
3.3.1 Custom Processor Architecture Implementation

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:



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

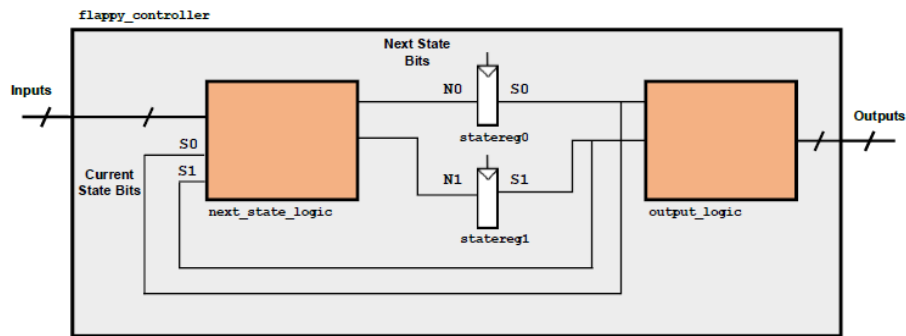


Figure 4: Complete Controller Template

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:

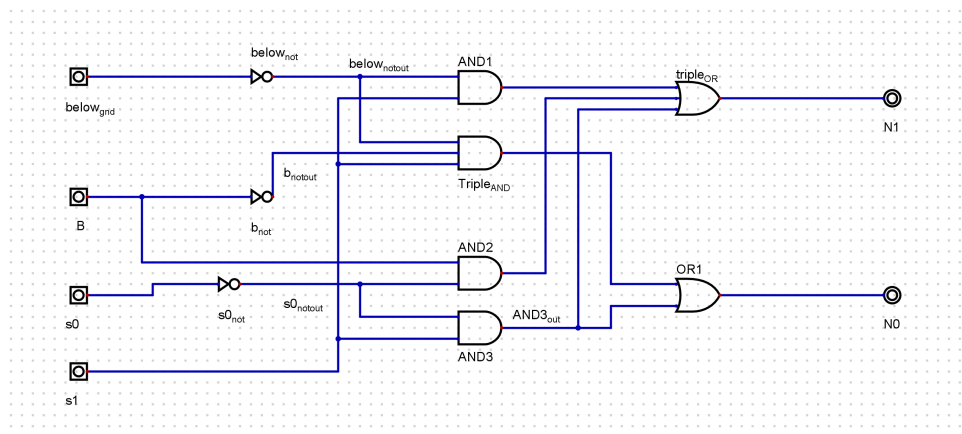


Figure 5: Next State Logic Schematic

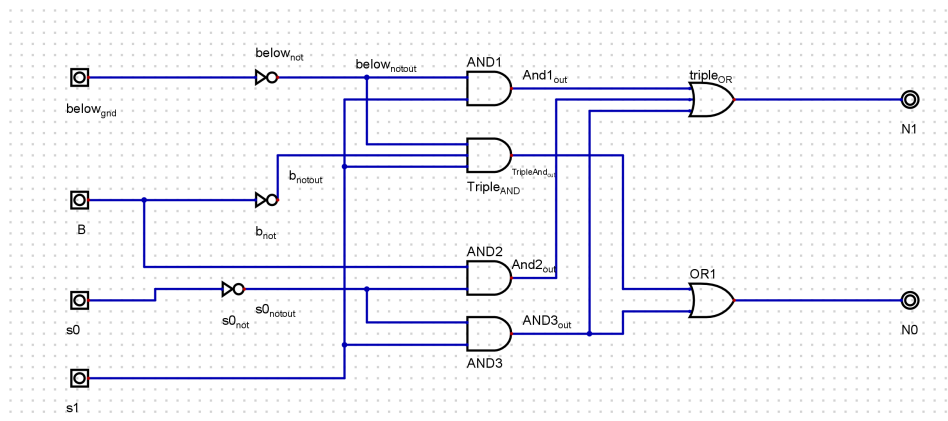


Figure 6: Output Logic Schematic

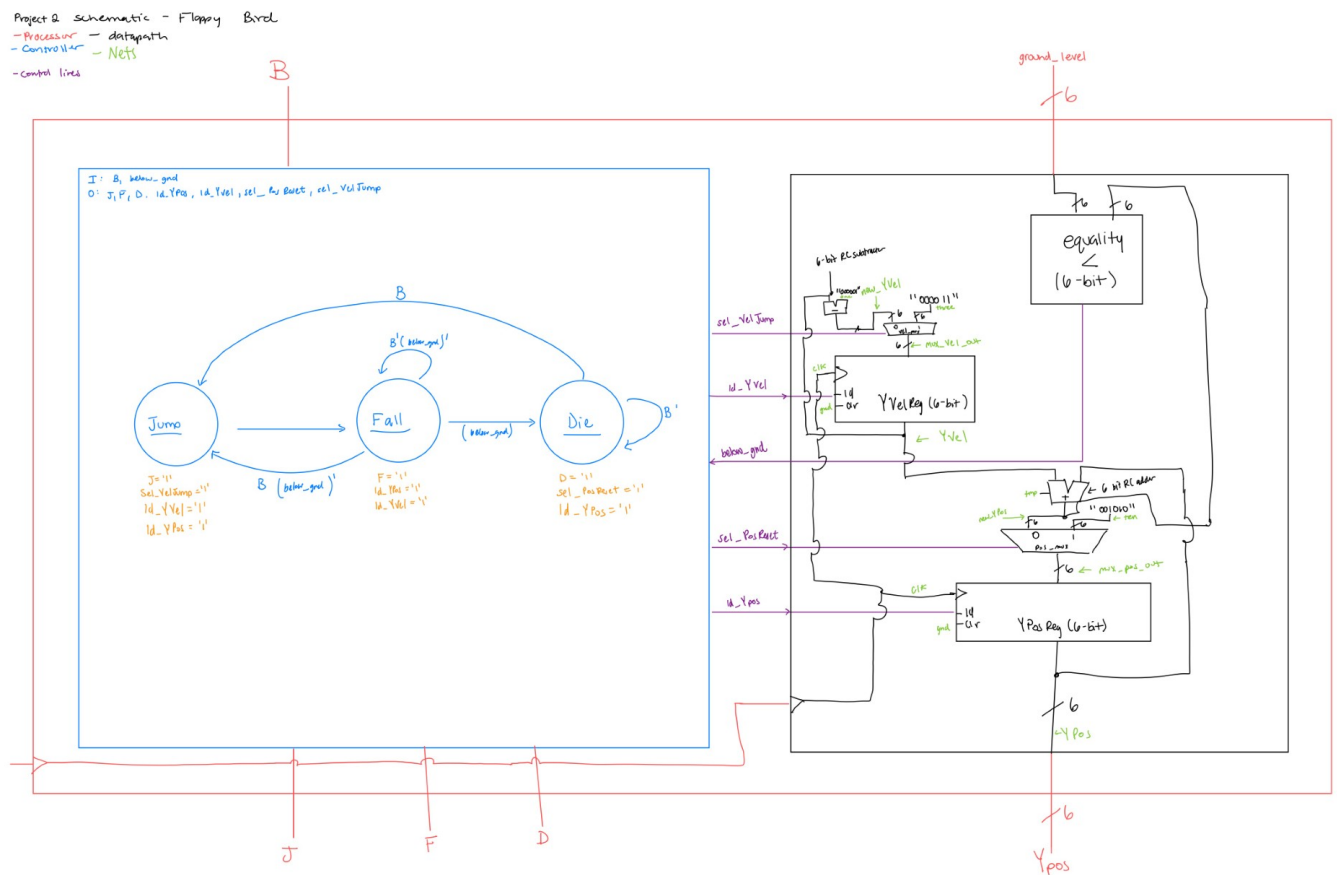


Figure 7: Complete Processor Schematic With Abstracted Controller

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

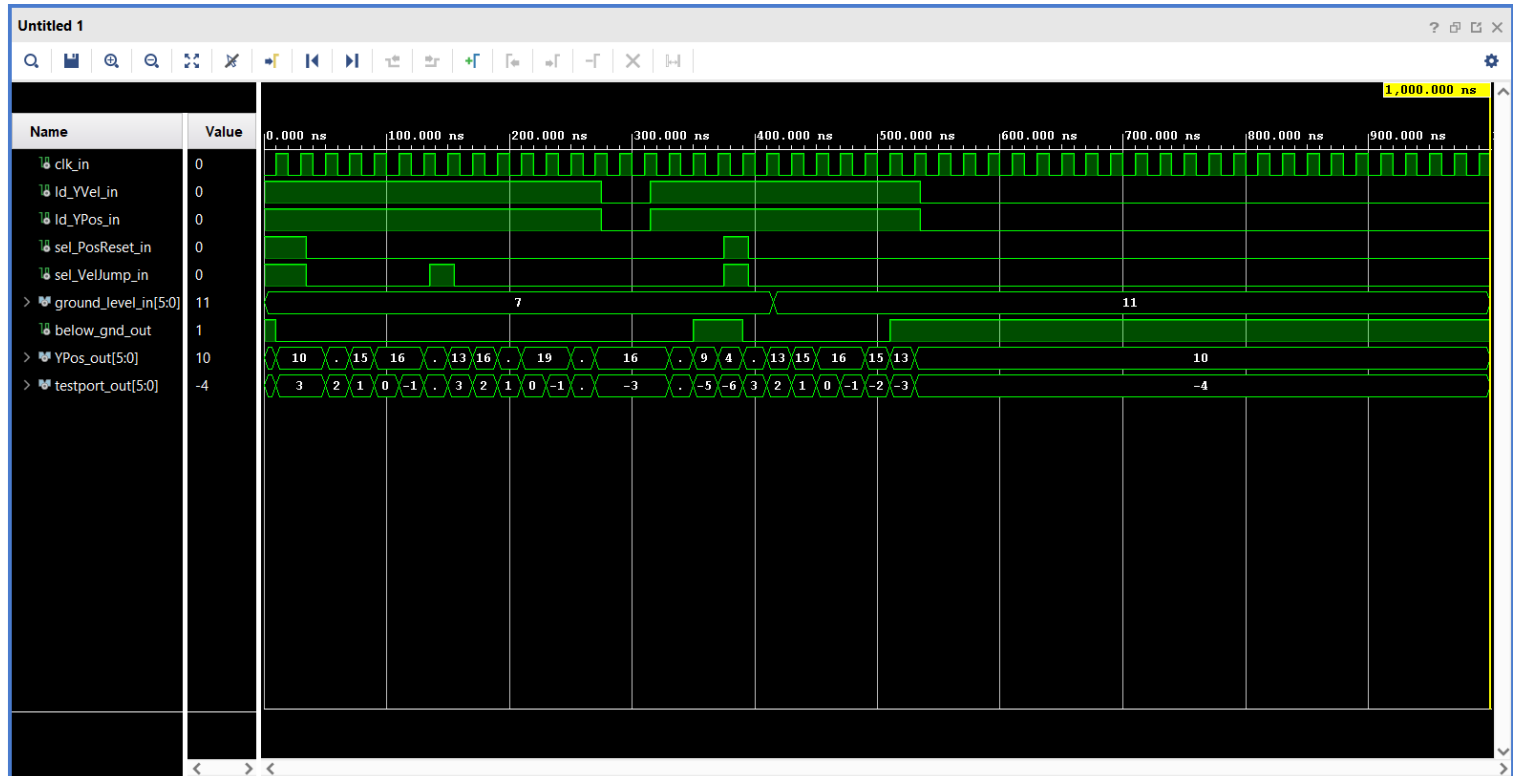


Figure 8: Datapath Testbench Simulation Results

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

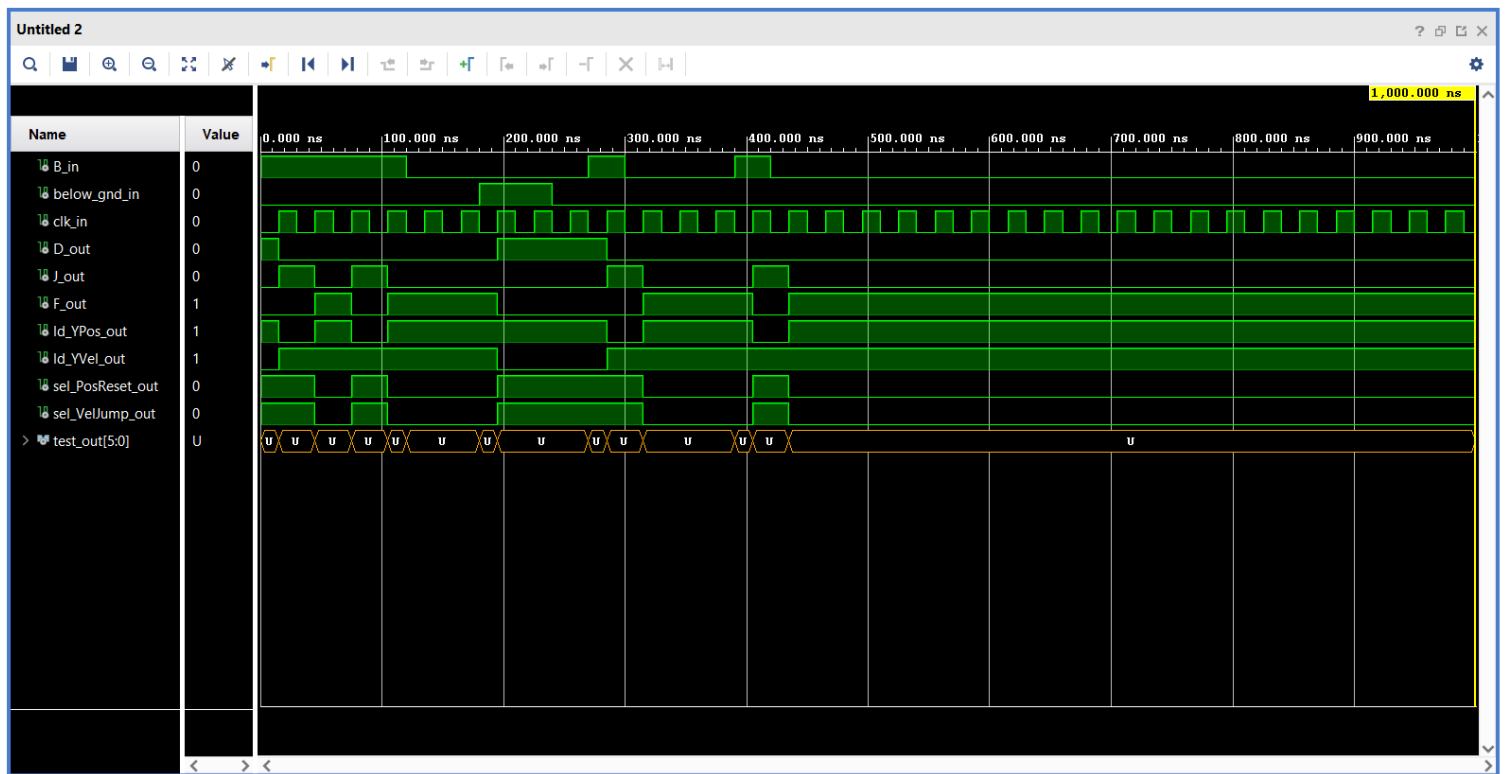


Figure 9: Controller Testbench Simulation Results

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

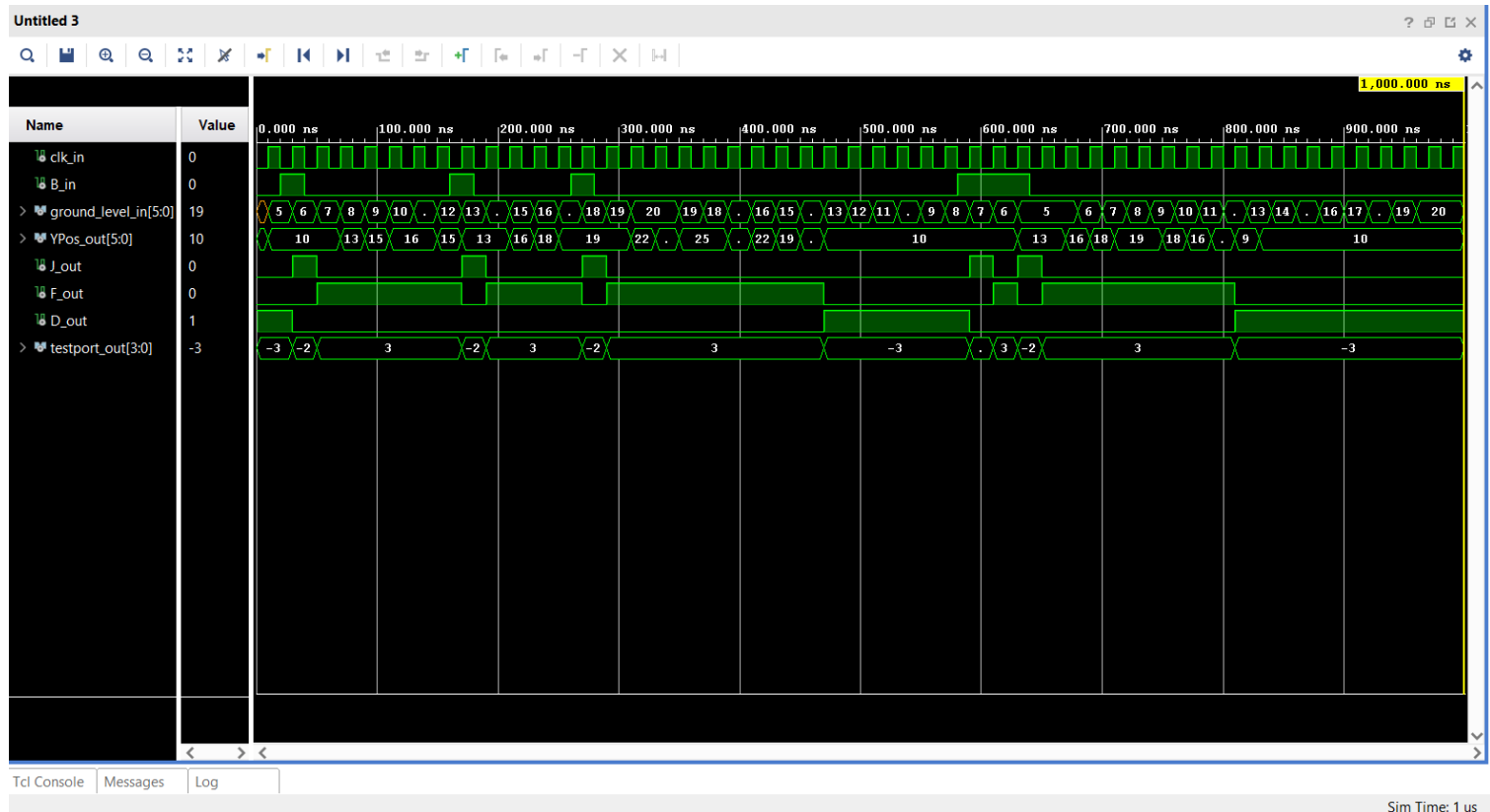


Figure 10: Complete Processor Testbench Simulation Results

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 HLMSM

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

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

```

1  -- Component to implement all necessary mathematical and multi-bit
2  -- operations
3  -- for the flappy game simulator project.
4
5  library IEEE;
6  use IEEE.STD_LOGIC_1164.ALL;
7
8  entity flappy_datapath is
9      Port (clk: in std_logic;
10           ld_YPos: in std_logic; -- Updates YPosReg when 1
11           ld_YVel: in std_logic; -- Updates YVelReg when 1
12           sel_PosReset: in std_logic; -- Select bit: 1 means reset YPos
13           to 10
14           sel_VelJump: in std_logic; -- Select bit: 1 means set YVel to 3
15           below_gnd: out std_logic; -- Character is below ground (YPos <
16           ground_level)
17           ground_level: in std_logic_vector (5 downto 0);
18           YPos: inout std_logic_vector (5 downto 0);
19           testport: out std_logic_vector (5 downto 0));
20 end flappy_datapath;
21
22 architecture structural of flappy_datapath is
23
24     -- Components we need (add below):
25     component lessthan_6bit
26     Port ( A : in STD_LOGIC_VECTOR (5 downto 0);
27           B : in STD_LOGIC_VECTOR (5 downto 0);
28           E : out STD_LOGIC);
29 end component;
30
31 component six_bit_load_register
32 Port ( ld : in STD_LOGIC;
33       clr : in STD_LOGIC;
34       clk : in STD_LOGIC;
35       D : in STD_LOGIC_VECTOR(5 downto 0); -- 6-bit data input
36       Q : out STD_LOGIC_VECTOR(5 downto 0)); -- 6-bit data output
37 end component;
38
39 component six_bit_ripple_carry_adder
40 Port ( X : in STD_LOGIC_VECTOR(5 downto 0);
41       Y : in STD_LOGIC_VECTOR(5 downto 0);
42       S : out STD_LOGIC_VECTOR(5 downto 0);
43       C_Out : out STD_LOGIC);
44 end component;

```

```

43 component six_bit_two_to_one_multiplexer
44     Port ( A_in : in STD_LOGIC_VECTOR(5 downto 0);
45           B_in : in STD_LOGIC_VECTOR(5 downto 0);
46           S_in : in STD_LOGIC;
47           Q_out : out STD_LOGIC_VECTOR(5 downto 0));
48 end component;
49
50 component six_bit_ripple_carry_subtractor
51     Port ( X : in STD_LOGIC_VECTOR(5 downto 0);
52           Y : in STD_LOGIC_VECTOR(5 downto 0);
53           S : out STD_LOGIC_VECTOR(5 downto 0);
54           C_Out : out STD_LOGIC);
55 end component;
56
57 -- hardcoded signals we need (add others as needed):
58 signal gnd: std_logic := '0'; -- to hardcode inputs to 0
59 signal tmp: std_logic; -- to be tied to unused outputs.
60 signal three: std_logic_vector(5 downto 0) := "000011"; -- hardcoded
    number three
61 signal one: std_logic_vector(5 downto 0) := "000001"; -- hardcoded
    number one
62 signal ten: std_logic_vector(5 downto 0) := "001010"; -- hardcoded
    number 10
63
64 -- Additional signals for internal nets
65 signal YVel: std_logic_vector (5 downto 0); -- signal to use as output
    of velocity register
66 -- Add other signals for internal nets as needed
67 signal new_YPos: std_logic_vector (5 downto 0); -- new position
    calculation
68 signal new_YVel: std_logic_vector (5 downto 0); -- new velocity
    calculation
69 signal mux_vel_out: std_logic_vector (5 downto 0); -- velocity mux
    output
70 signal mux_pos_out: std_logic_vector (5 downto 0); -- position mux
    output
71
72
73 begin
74
75 -- Add instances of components here
76 vel_reg: six_bit_load_register port map (ld => ld_YVel, clr => gnd, clk
    => clk, D => mux_vel_out, Q => YVel);
77 pos_reg: six_bit_load_register port map (ld => ld_YPos, clr => gnd, clk
    => clk, D => mux_pos_out, Q => YPos);
78 equality: lessthan_6bit port map (A => new_YPos, B => ground_level, E
    => below_gnd);
79 adder: six_bit_ripple_carry_adder port map (X => YVel, Y => YPos, S =>
    new_YPos, C_out => tmp);
80 subtractor: six_bit_ripple_carry_subtractor port map ( X => YVel, Y =>
    one, S => new_YVel, C_out => tmp);
81 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);
82 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);
83
84 -- The testport can be used for evaluating internal nets.
85 --     Test a bus

```

```

86 testport <= YVel; -- default . . . output velocity value.
87
88 -- Test individual bits
89 -- testport(0) <= gnd;
90
91 end structural;

```

6.3 Controller VHDL Code (Task 3)

Figure A2: flappy_controller.vhd Implementation

```

1  -- Flappy controller consisting of:
2  -- 1) Next state logic
3  -- 2) State registers (flip flops)
4  -- 3) Output logic
5
6  -- This file should NOT need to be modified by students. Rather they
7  -- should update the
8  -- next state and output modules referenced here.
9
10 library IEEE;
11 use IEEE.STD_LOGIC_1164.ALL;
12
13 entity flappy_controller is
14     Port ( CLK : in STD_LOGIC;
15           B : in STD_LOGIC;
16           below_gnd : in STD_LOGIC;
17           J : out STD_LOGIC;
18           F : out STD_LOGIC;
19           D : out STD_LOGIC;
20           ld_YPos : out STD_LOGIC;
21           ld_YVel : out STD_LOGIC;
22           sel_PosReset: out std_logic;
23           sel_VelJump: out std_logic;
24           testport : out std_logic_vector (5 downto 0));
25 end flappy_controller;
26
27 architecture Structural of flappy_controller is
28
29     component flappy_output_logic is
30         Port ( S0: in std_logic;
31               S1: in std_logic;
32               J : out STD_LOGIC;
33               F : out STD_LOGIC;
34               D : out STD_LOGIC;
35               ld_YPos : out STD_LOGIC;
36               ld_YVel : out STD_LOGIC;
37               sel_PosReset: out std_logic;
38               sel_VelJump: out std_logic;
39               testport : out std_logic);
40     end component;
41
42     component flappy_nextstate_logic is
43         Port ( B : in STD_LOGIC;
44               below_gnd : in STD_LOGIC;

```

```

44         S0 : in STD_LOGIC;
45         S1 : in STD_LOGIC;
46         N0 : out STD_LOGIC;
47         N1 : out STD_LOGIC;
48         testport : out std_logic);
49 end component;
50
51 component dflipflop is
52     Port ( D : in STD_LOGIC;
53           CLK : in STD_LOGIC;
54           LD : in std_logic ;
55           CLR : in std_logic;
56           Q : inout STD_LOGIC := '0');
57 end component;
58
59 signal S0, S1, N0, N1: std_logic;
60 signal vdd : std_logic := '1';
61 signal gnd : std_logic := '0';
62 signal next_test, output_test : std_logic;
63
64 begin
65
66 -- State Registers
67 statereg0: dflipflop port map (N0, CLK, vdd, gnd, S0);
68 statereg1: dflipflop port map (N1, CLK, vdd, gnd, S1);
69
70 --Next State Logic
71 nextstate: flappy_nextstate_logic port map (B, below_gnd, S0, S1, N0,
72       N1, next_test);
73
74 --Output Logic
75 output: flappy_output_logic port map ( S0, S1, J, F, D, ld_YPos,
76       ld_YVel, sel_PosReset, sel_VelJump, output_test);
77
78 --testport debug (modify as needed)
79 testport(5) <= S0;
80 testport(4) <= S1;
81 testport(3) <= N0;
82 testport(2) <= N1;
83
84 end Structural;

```

6.4 Processor VHDL Code (Task 4)

Figure A3: flappy_processor.vhd Implementation

```

1  -- Highest level of flappy game simulator.  Connections between
2  controller and datapath
3
4  library IEEE;
5  use IEEE.STD_LOGIC_1164.ALL;
6
7  entity flappy_processor is
8      Port ( CLK : in STD_LOGIC ;
9            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,
47        below_gnd_net : std_logic;
48 signal control_test, data_test : std_logic_vector (5 downto 0);
49
50 begin
51 controller: flappy_controller port map (CLK, B, below_gnd_net, J, F, D,
52        ld_YPos_net, ld_YVel_net,
53        sel_PosReset_net,
54        sel_VelJump_net, control_test);
55 datapath: flappy_datapath port map (CLK, ld_YPos_net, ld_YVel_net,
56        sel_PosReset_net, sel_VelJump_net, below_gnd_net,
57        ground_level, YPos, data_test);
58
59 -- Debug port (can be modified by students)
60 testport(0) <= ld_YPos_net;
61 testport(1) <= ld_YVel_net;
62 testport(2) <= sel_PosReset_net;
63 testport(3) <= sel_VelJump_net;
64
65 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.