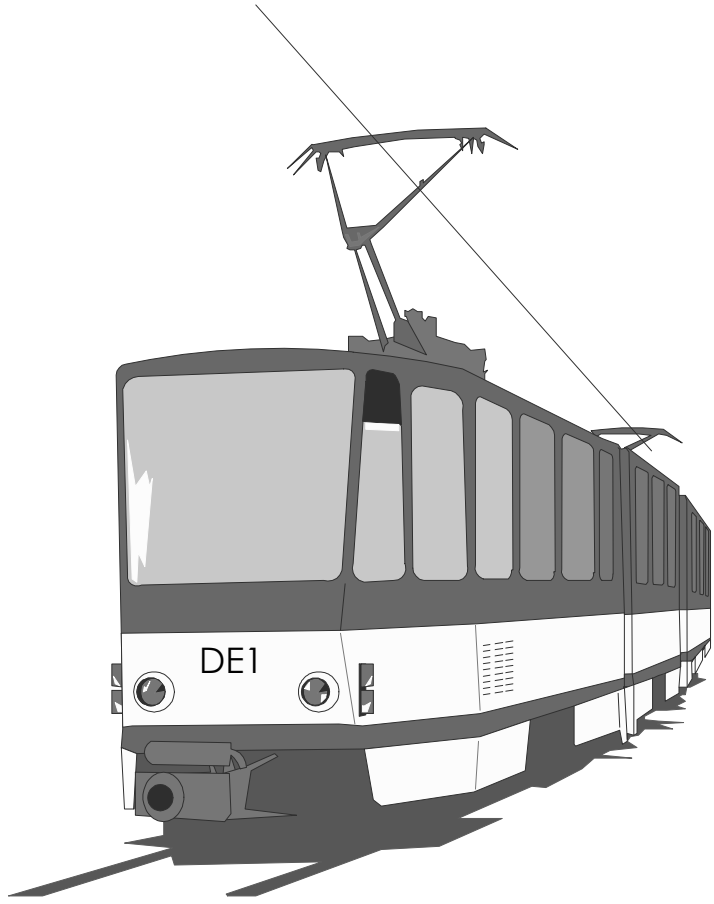
CHAPTER 8

State Machine Design: The Electric Train Controller



8 State Machine Design: The Electric Train Controller

8.1 The Train Control Problem

The track layout of a small electric train system is shown in Figure 8.1. Two trains, we'll call A and B, run on the tracks, hopefully without colliding. To avoid collisions, the trains require a safety controller that allows trains to move in and out of intersections without mishap.

For safe operation, only one train at a time can be present on any given track segment. The track layout seen in Figure 8.1 is divided into four track segments. Each track segment has sensors that are used to detect trains at the entry and exit points.

In Figure 8.1, there are two Trains A and B. As an example, assume Train A always runs on the outer track loop and Train B on the inner track loop. Assume for a moment that Train A has just passed Sensor 4 and is near Switch 3 moving counterclockwise. Let's also assume that Train B is moving counterclockwise and approaching Sensor 2. Since Train B is entering the common track (Track 2), Train A must be stopped when it reaches Sensor 1, and must wait until Train B has passed Sensor 3 (i.e., Train B is out of the common track). At this point, the track switches should switch for Train A, Train A will be allowed to enter Track 2, and Train B will continue moving toward Sensor 2.

The controller is a state machine that uses the sensors as inputs. The controller's outputs control the direction of the trains and the position of the switches. However, the state machine does not control the speed of the train. This means that the system controller must function correctly independent of the speed of the two trains.

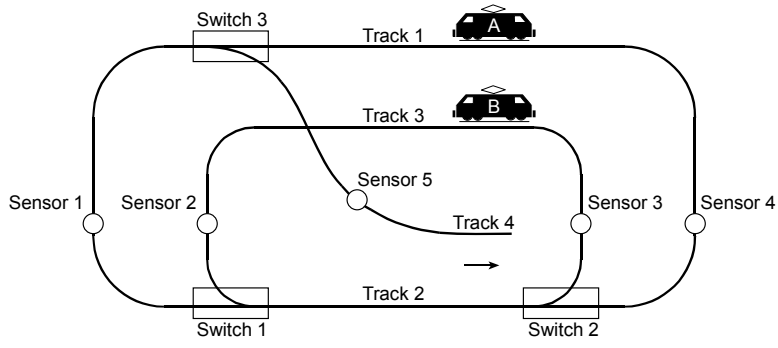


Figure 8.1 Track layout with input sensors and output switches and output tracks.

An FPGA-based "virtual" train simulation will be used that emulates this setup and provides video output. Since there are no actual power circuits connected to a train on the FPGA board, it is only intended to give you a visual indication of how the output signals work in the real system. The following sections describe how the state machine should control each signal to operate the trains properly.

8.2 Train Direction Outputs (DA1-DA0, and DB1-DB0)

The direction for each train is controlled by four output signals (two for each train), DA (DA1-DA0) for train A, and DB (DB1-DB0) for train B[2]. When these signals indicate forward "01" for a particular train, a train will move counterclockwise (on track 4, the train moves toward the outer track). When the signals imply reverse "10", the train(s) will move clockwise. The "11" value is illegal and should not be used. When these signals are set to "00", a train will stop. (See Figure 8.2.)

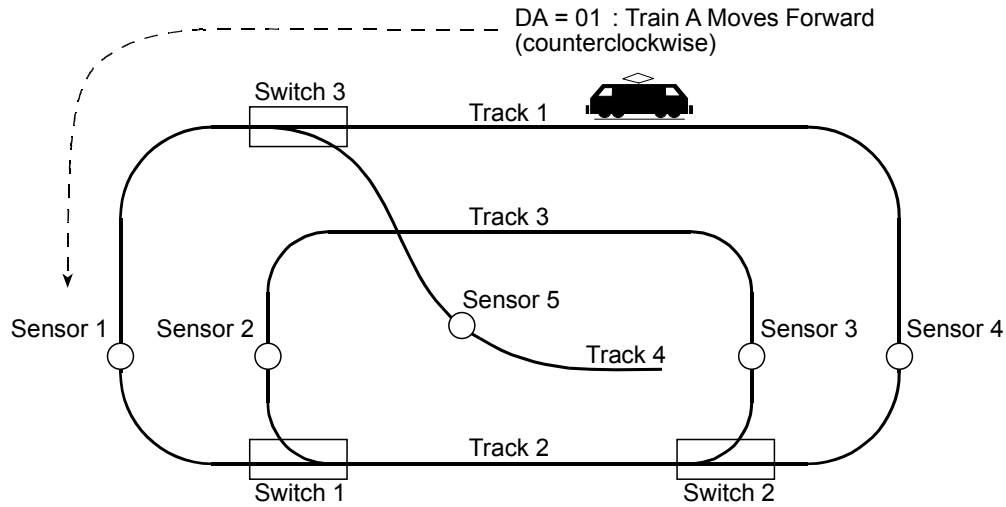


Figure 8.2 Controlling the train's motion with the train direction signals.

[2] For those familiar with earlier editions of this book, additional track power signals were required for power control relays. This new train problem is based on newer digital DCC model trains and it no longer needs the track power signals and relays, so they have been eliminated. The signals work exactly the same as the previous train setup, if you assume that track power supply A always runs train A and track power supply B always runs train B.

8.3 Switch Direction Outputs (SW1, SW2, and SW3)

Switch directions are controlled by asserting the SW1, SW2, and SW3 output signals either high (outside connected with inside track) or low (outside tracks connected). That is, anytime all of the switches are set to 1, the tracks are setup such that the outside tracks are connected to the inside tracks. (See Figure 8.3.)

If a train moves the wrong direction through an open switch it will derail. Be careful. If a train is at the point labeled "Track 1" in Figure 8.3 and is moving to the left, it will derail at Switch 3. To keep it from derailing, SW3 would need to be set to 0.

Also, note that Tracks 3 and 4 cross at an intersection and care must be taken to avoid a crash at this point.

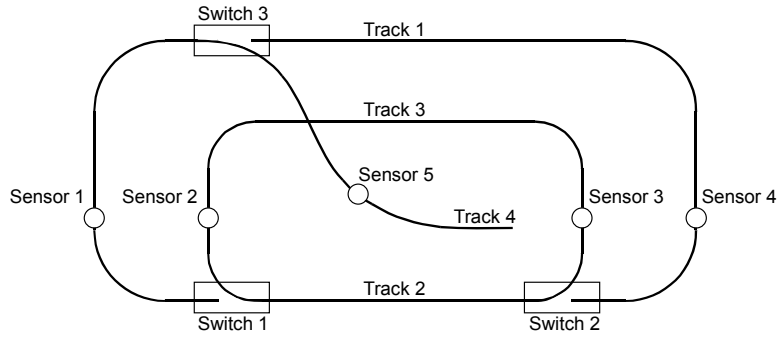


Figure 8.3 Track direction if all switches are asserted (SW1 = SW2 = SW3 = 1)

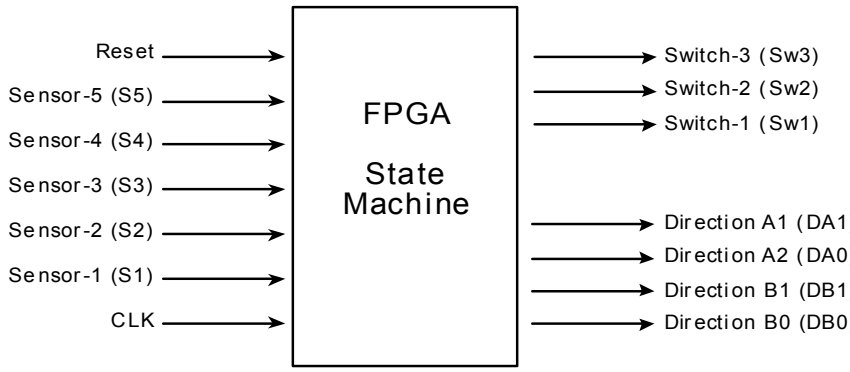
8.4 Train Sensor Input Signals (S1, S2, S3, S4, and S5)

The five train sensor input signals (S1, S2, S3, S4, and S5) go high when a train is near the sensor location. It should be noted that sensors (S1, S2, S3, S4, and S5) *do not go high for only one clock cycle*. In fact, the sensors fire continuously for *many* clock cycles per passage of a train. This means that if your design is testing the same sensor from one state to another, you must wait for the signal to change from high to low.

As an example, if you wanted to count how many times that a train passes Sensor 1, you can not just have an "IF S1 GOTO count-one state" followed by "IF S1 GOTO count-two state." You would need to have a state that sees S1=‘1’, then S1=‘0’, then S1=‘1’ again before you can be sure that it has passed S1 twice. If your state machine has two concurrent states that look for S1=’1’, the state machine will pass through both states in two consecutive clock cycles although the train will have passed S1 only once.

Another way would be to detect S1=’1’, then S4=’1’, then S1=’1’ if, in fact, the train was traversing the outside loop continuously. Either method will ensure that the train passed S1 twice.

The state machine's signal inputs and outputs have been summarized in the following figure:



Sensor (S1, S2, S3, S4, S5) = 1 Train Present

= 0 Train not Present

Switches (SW1, SW2, SW3) = 0 Connected to Outside Track

= 1 Connected to Inside Track

Train Direction (DA1-DA0) and (DB1-DB0) = 00 Stop

= 01 Forward (Counterclockwise)

= 10 Backward (Clockwise)

Figure 8.4 Train Control State Machine I/O Configuration.

8.5 An Example Controller Design

We will now examine a working example of a train controller state machine. For this controller, two trains run counterclockwise at various speeds and avoid collisions. One Train (A) runs on the outer track and the other train (B) runs on the inner track. Only one train at a time is allowed to occupy the common track.

Both an ASM chart and a classic state bubble diagram are illustrated in Figures 8.5 and 8.6 respectively. In the ASM chart, state names, ABout, Ain, Bin, Bstop, and Astop indicate the active and possible states. The rectangles contain

the active (High) outputs for the given state. Outputs not listed are always assumed to be inactive (Low).

The diamond shapes in the ASM chart indicate where the state machine tests the condition of the inputs (S1, S2, etc.). When two signals are shown in a diamond, they are both tested at the same time for the indicated values.

A state machine classic bubble diagram is shown in Figure 8.6. Both Figures 8.5 and 8.6 contain the same information. They are simply different styles of representing a state diagram. The track diagrams in Figure 8.7 show the states visually. In the state names, "in" and "out" refer to the state of track 2, the track that is common to both loops.

Description of States in Example State Machine

All States

- All signals that are not "Asserted" are zero and imply a logical result as described.

ABout: "Trains A and B Outside"

- DA0 Asserted: Train A is on the outside track and moving counterclockwise (forward).
- DB0 Asserted: Train B is on the inner track (not the common track) and also moving forward.
- Note that by NOT Asserting DA1, it is automatically zero -- same for DB1. Hence, the outputs are DA = “01” and DB = “01”.

Ain: "Train A moves to Common Track"

- Sensor 1 has fired either first or at the same time as Sensor 2.
- Either Train A is trying to move towards the common track, or
- Both trains are attempting to move towards the common track.
- Both trains are allowed to enter here; however, state Bstop will stop B if both have entered.
- DA0 Asserted: Train A is on the outside track and moving counterclockwise (forward).
- DB0 Asserted: Train B is on the inner track (not the common track) and also moving forward.

Bstop: "Train B stopped at S2 waiting for Train A to clear common track"

- DA0 Asserted: Train A is moving from the outside track to the common track.
- Train B has arrived at Sensor 2 and is stopped and waits until Sensor 4 fires.
- SW1 and SW2 are NOT Asserted to allow the outside track to connect to common track.

Bin: "Train B has reached Sensor 2 before Train A reaches Sensor 1"

- Train B is allowed to enter the common track. Train A is approaching Sensor 1.
- DA0 Asserted: Train A is on the outside track and moving counterclockwise (forward).
- DB0 Asserted: Train B is on the inner track moving towards the common track.
- SW1 Asserted: Switch 1 is set to let the inner track connect to the common track.
- SW2 Asserted: Switch 2 is set to let the inner track connect to the common track.

Astop: "Train A stopped at S1 waiting for Train B to clear the common track"

- DB0 Asserted: Train B is on the inner track moving towards the common track.
- SW1 and SW2 Asserted: Switches 1 and 2 are set to connect the inner track to the common track.

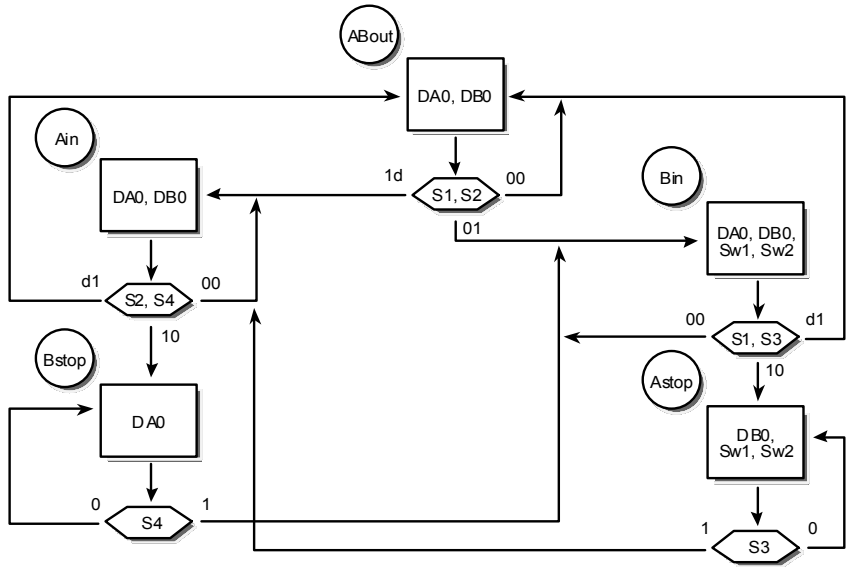


Figure 8.5 Example Train Controller ASM Chart.

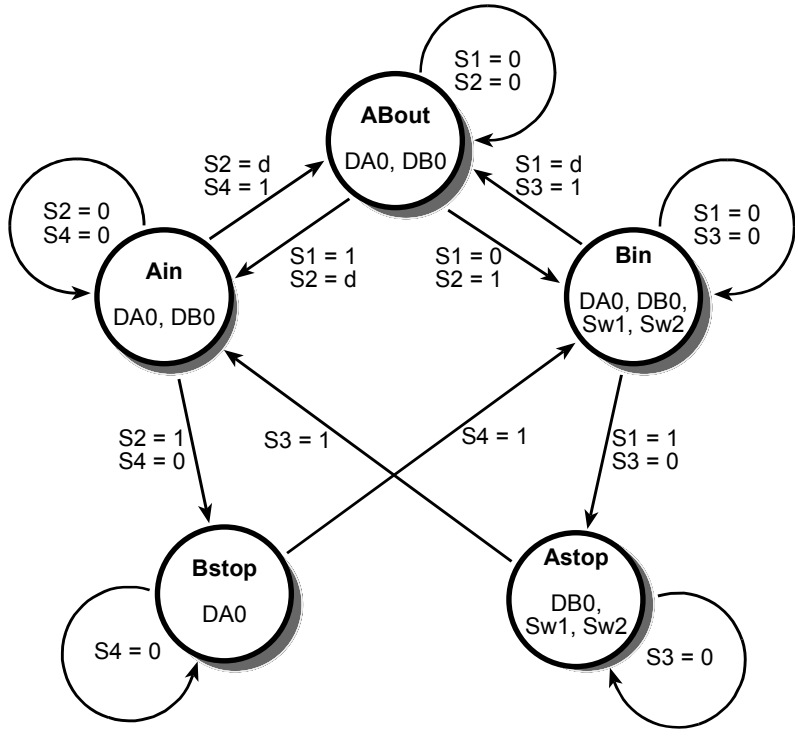


Figure 8.6 Example Train Controller State Diagram.

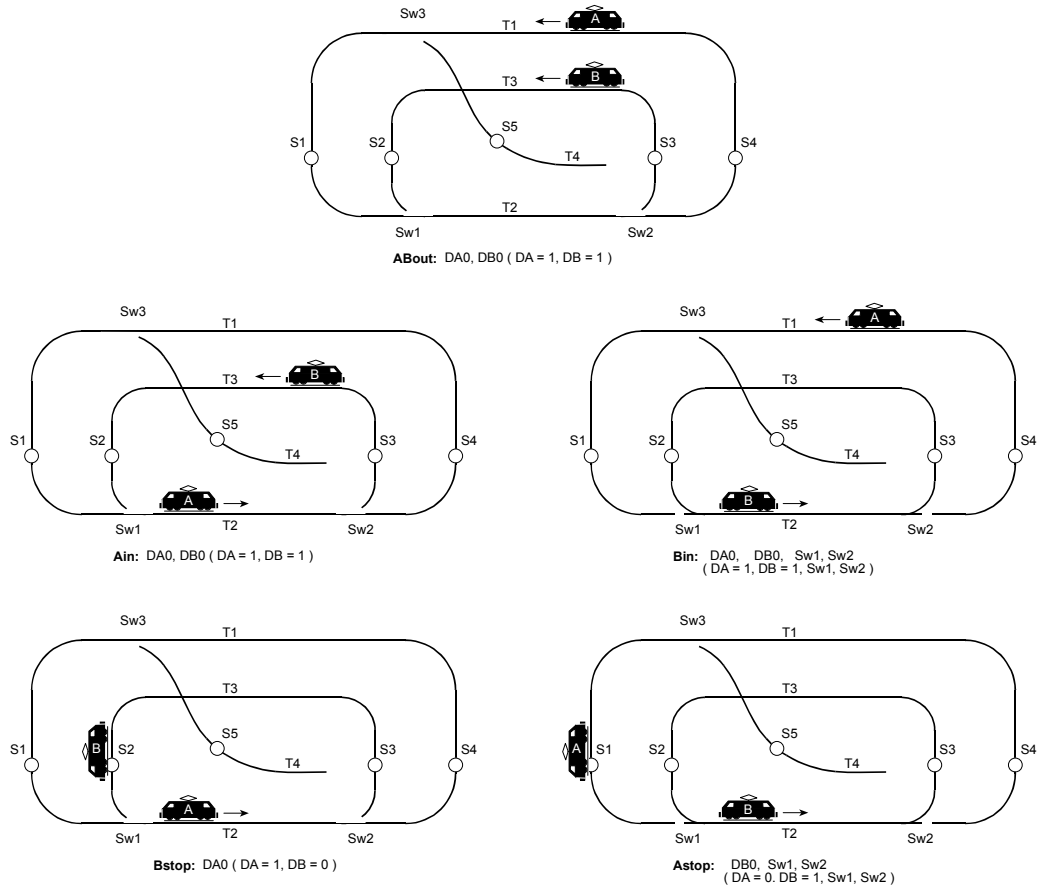


Figure 8.7 Working diagrams of train positions for each state.

Table 8.1 Outputs corresponding to states.

State	ABout	Ain	Astop	Bin	Bstop
Sw1	0	0	1	1	0
Sw2	0	0	1	1	0
Sw3	0	0	0	0	0
DA(1-0)	01	01	00	01	01
DB(1-0)	01	01	01	01	00

8.6 VHDL Based Example Controller Design

The corresponding VHDL code for the state machine in Figures 8.5 and 8.6 is shown below. A CASE statement based on the current state examines the inputs to select the next state. At each clock edge, the next state becomes the current state. WITH…SELECT statements at the end of the program specify the

outputs for each state. For additional VHDL help, see the help files in the Altera CAD tools or look at the VHDL examples in Chapter 6.

```
-- Example State machine to control trains-- File: Tcontrol.vhd
--
-- These libraries are required in all VHDL source files
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.ALL;
USE  IEEE.STD_LOGIC_ARITH.ALL;
USE  IEEE.STD_LOGIC_UNSIGNED.ALL;

                -- This section defines state machine inputs and outputs
                -- No modifications should be needed in this section
ENTITY Tcontrol IS
PORT(  reset, clock, sensor1, sensor2,
        sensor3, sensor4, sensor5       : IN      STD_LOGIC;
        switch1, switch2, switch3       : OUT     STD_LOGIC;
                -- dirA and dirB are 2-bit logic vectors(i.e. an array of 2 bits)
       dirA, dirB                       : OUT STD_LOGIC_VECTOR( 1 DOWNTO 0 ));
END Tcontrol;

                        -- This code describes how the state machine operates
                        -- This section will need changes for a different  state machine
ARCHITECTURE a OF Tcontrol IS

                        -- Define local signals (i.e. non input or output signals) here
    TYPE STATE_TYPE IS ( ABout, Ain, Bin, Astop, Bstop );
    SIGNAL state: STATE_TYPE;
    SIGNAL sensor12, sensor13, sensor24   : STD_LOGIC_VECTOR(1 DOWNTO 0);

BEGIN
                        -- This section describes how the state machine behaves
                        -- this process runs once every time reset or the clock changes
    PROCESS ( reset, clock )
    BEGIN
                        -- Reset to this state (i.e. asynchronous reset)
        IF reset = '1' THEN
                state <= ABout;
        ELSIF clock'EVENT AND clock = '1' THEN

                        -- clock'EVENT means value of clock just changed
                        --This section will execute once on each positive clock edge
                        --Signal assignments in this section will generate D flip-flops
                        -- Case statement to determine next state
                CASE state IS
                        WHEN ABout =>
                                -- This Case checks both sensor1 and sensor2 bits
                           CASE Sensor12 IS
                                -- Note: VHDL's use of double quote for bit vector versus
                                -- a single quote for only one bit!
                                WHEN "00" => state <= About;
                                WHEN "01" => state <= Bin;
```

```
                        WHEN "10" => state <= Ain;
                        WHEN "11" => state <= Ain;
                        -- Default case is always required
                        WHEN OTHERS => state <= ABout;
                    END CASE;

                WHEN Ain =>
                    CASE Sensor24 IS
                        WHEN "00" => state <= Ain;
                        WHEN "01" => state <= ABout;
                        WHEN "10" => state <= Bstop;
                        WHEN "11" => state <= ABout;
                        WHEN OTHERS => state <= ABout;
                    END CASE;

                WHEN Bin =>
                    CASE Sensor13 IS
                        WHEN "00" => state <= Bin;
                        WHEN "01" => state <= ABout;
                        WHEN "10" => state <= Astop;
                        WHEN "11" => state <= About;
                        WHEN OTHERS => state <= ABout;
                    END CASE;

                WHEN Astop =>
                    IF Sensor3 = '1' THEN
                        state <= Ain;
                    ELSE
                        state <= Astop;
                    END IF;

                WHEN Bstop =>
                    IF Sensor4 = '1' THEN
                        state <= Bin;
                    ELSE
                        state <= Bstop;
                    END IF;
            END CASE;
    END IF;
END PROCESS;
                        -- combine sensor bits for case statements above
                        -- "&" operator combines bits
sensor12 <= sensor1 & sensor2;
sensor13 <= sensor1 & sensor3;
sensor24 <= sensor2 & sensor4;

                        -- These outputs do not depend on the state
   Switch3 <= '0';
```

```
                    -- Outputs that depend on state, use state to select value
                    -- Be sure to specify every output for every state
                    -- values will not default to zero!
    WITH state SELECT
            Switch1   <=      '0'     WHEN ABout,
                              '0'     WHEN Ain,
                              '1'     WHEN Bin,
                              '1'     WHEN Astop,
                              '0'     WHEN Bstop;
    WITH state SELECT
            Switch2    <=     '0'     WHEN ABout,
                              '0'     WHEN Ain,
                              '1'     WHEN Bin,
                              '1'     WHEN Astop,
                              '0'      WHEN Bstop;
    WITH state SELECT
            DirA      <=      "01"    WHEN ABout,
                              "01"    WHEN Ain,
                              "01"    WHEN Bin,
                              "00"    WHEN Astop,
                              "01"    WHEN Bstop;
    WITH state SELECT
            DirB      <=      "01"    WHEN ABout,
                              "01"    WHEN Ain,
                              "01"    WHEN Bin,
                              "01"    WHEN Astop,
                              "00"    WHEN Bstop;
END a;
```

8.7 Verilog Based Example Controller Design

The corresponding Verilog code for the state machine in Figures 8.5 and 8.6 is shown below. A CASE statement based on the current state examines the inputs to select the next state. At each clock edge, the next state becomes the current state. A second CASE statement at the end of the program specifies the outputs for each state. For additional Verilog help, see the help files in the Altera CAD tools or look at the Verilog examples in Chapter 7.

```
                    // Example Verilog State machine to control trains
module Tcontrol (reset, clock, sensor1, sensor2, sensor3, sensor4, sensor5,
                    switch1, switch2, switch3, dirA, dirB);
                    // This section defines state machine inputs and outputs
                    // No modifications should be needed in this section
input reset, clock, sensor1, sensor2, sensor3, sensor4, sensor5;
output switch1, switch2, switch3;
output [1:0] dirA, dirB;
reg switch1, switch2;
                    // dirA and dirB are 2-bit logic vectors(i.e. an array of 2 bits)
reg [1:0] dirA, dirB;
reg [2:0] state;
```

```
                    // This code describes how the state machine operates
                    // This section will need changes for a different state machine
                    // State assignments are needed in Verilog
parameter ABout = 0, Ain = 1, Bin = 2, Astop = 3, Bstop = 4;
                    // This section describes how the state machine behaves
                    // this process runs once every time reset or the clock changes
always @(posedge clock or posedge reset)
        begin
                    // Reset to this state (i.e. asynchronous reset)
                if (reset)
                    state = ABout;
                else
                    // posedge clock means positive clock edge
                    //This section will execute once on each positive clock edge
                    //Signal assignments in this section will generate D flip-flops
                case (state)    // Case statement to determine next state
                    ABout:
                        // This Case checks both sensor1 and sensor2 bits
                        case (sensor12)
                            2'b 00: state = ABout;
                            2'b 01: state = Bin;
                            2'b 10: state = Ain;
                            2'b 11: state = Ain;
                            // Default case is needed here
                            default: state = ABout;
                        endcase
                    Ain:
                        case (sensor24)
                            2'b 00: state = Ain;
                            2'b 01: state = ABout;
                            2'b 10: state = Bstop;
                            2'b 11: state = ABout;
                            default: state = ABout;
                        endcase
                    Bin:
                        case (sensor13)
                            2'b 00: state = Bin;
                            2'b 01: state = ABout;
                            2'b 10: state = Astop;
                            2'b 11: state = ABout;
                            default: state = ABout;
                        endcase
                    Astop:
                        if (sensor3)
                            state = Ain;
                        else
                            state = Astop;
                    Bstop:
                        if (sensor4)
                            state = Bin;
                        else
                            state = Bstop;
```

```
                default: state = ABout;
            endcase
        end
                // combine sensor bits for case statements above
                // { } operators combine bits
wire [1:0] sensor12 = {sensor1, sensor2};
wire [1:0] sensor13 = {sensor1, sensor3};
wire [1:0] sensor24 = {sensor2, sensor4};
                // These outputs do not depend on the state
wire switch3 = 0;
                // Outputs that depend on state, use state to select value
                // Be sure to specify every output for every state
                // values will not default to zero!
always @(state)
        begin
            case (state)
                ABout:
                    begin
                        switch1 = 0;
                        switch2 = 0;
                        dirA = 2'b 01;
                        dirB = 2'b 01;
                    end
                Ain:
                    begin
                        switch1 = 0;
                        switch2 = 0;
                        dirA = 2'b 01;
                        dirB = 2'b 01;
                    end
                Bin:
                    begin
                        switch1 = 1;
                        switch2 = 1;
                        dirA = 2'b 01;
                        dirB = 2'b 01;
                    end
                Astop:
                    begin
                        switch1 = 1;
                        switch2 = 1;
                        dirA = 2'b 00;
                        dirB = 2'b 01;
                    end
                Bstop:
                    begin
                        switch1 = 0;
                        switch2 = 0;
                        dirA = 2'b 01;
                        dirB = 2'b 00;
                    end
```

```
                    default:
                        begin
                            switch1 = 0;
                            switch2 = 0;
                            dirA = 2'b 00;
                            dirB = 2'b 00;
                        end
                endcase
            end
endmodule
```

8.8 Automatically Generating a State Diagram of a Design

You can use **Tools ⇨Netlist Viewers ⇨State Diagram Viewer** to automatically generate a state diagram and state table of a VHDL or Verilog based state machine after it has been compiled successfully as seen in Figure 8.8. The encoding tab at the bottom will also display the state encodings which typically use the one-hot encoding scheme (i.e., one flip-flop is used per state and the active flip-flop indicates the current state).

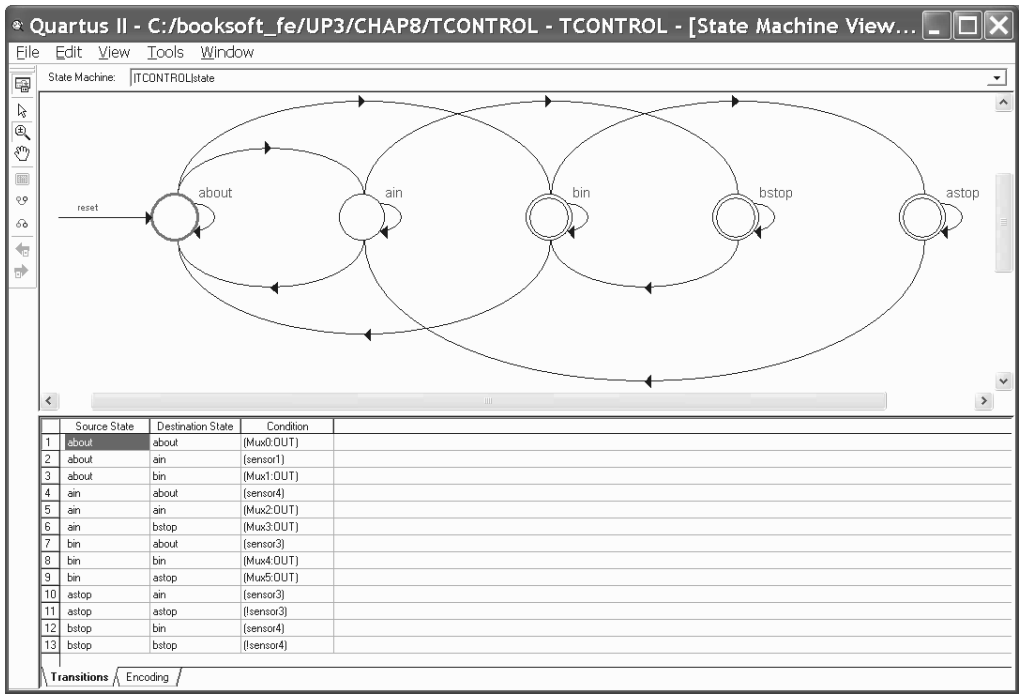


Figure 8.8 Automatically generated state diagram of Tcontrol.vhd.

8.9 Simulation Vector file for State Machine Simulation

The vector waveform file, *tcontrol.vwf*, seen in Figure 8.9 controls the simulation and tests the state machine. A vector waveform file specifies the simulation stimulus and display. This file sets up a 40ns clock and specifies sensor patterns (inputs to the state machine), which will be used to test the state machine. These patterns were chosen by picking a path in the state diagram that moves to all of the different states.

The sensor-input patterns will need to be changed if you change to a different train pattern, and therefore, the state machine. Sensor inputs should not change faster than the clock cycle time of 40ns. As a minimum, try to test all of the states and arcs in your state machine simulation.

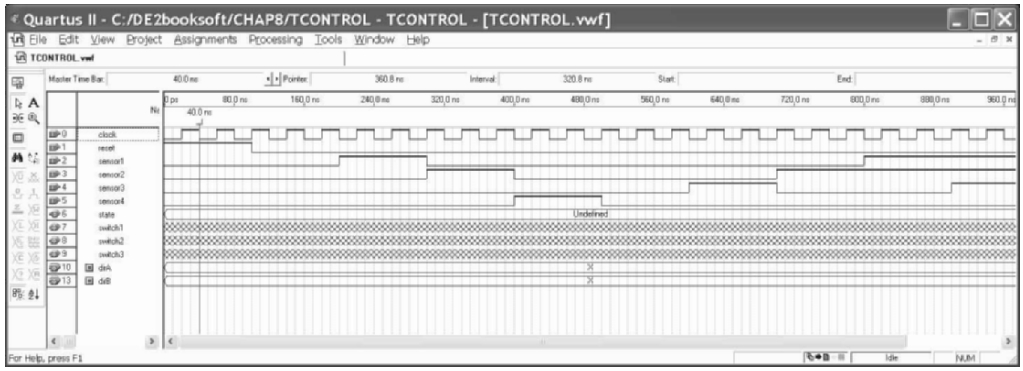


Figure 8.9 Tcontrol.vwf vector waveform file for simulation.

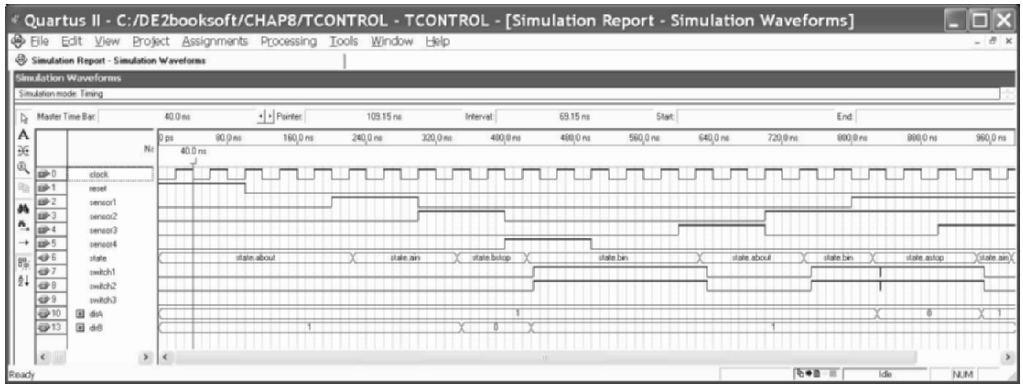


Figure 8.10 Simulation of Tcontrol.vhd using the Tcontrol.vwf vector waveform file in Figure 8.9.

8.10 Running the Train Control Simulation

Follow these steps to compile and simulate the state machine for the electric train controller.

Select Current Project

Make Tcontrol.vhd the current project with **File ⇨Open Project ⇨*Name***

Then find and select **Tcontrol.vhd**.

Compile and Simulate

Select **Processing ⇨Start Compilation and Simulation**. The simulator will run automatically if there are no compile errors. Select **Processing ⇨Simulation Report** to see the timing diagram display of your simulation as seen in Figure 8.10. Whenever you change your VHDL (or Verilog) source you will need to repeat this step. If you get compile errors, clicking on the error will move the text editor to the error location. The Altera software has extensive online help including HDL syntax examples.

Make any text changes to Tcontrol.vhd or Tcontrol.vwf (test vector waveform file) with **File ⇨Open**. This brings up a special editor window. Note that the menus at the top of the screen change depending on which window is currently open.

Updating new Simulation Test Vectors

To update the simulation with new test vectors from a modified Tcontrol.vwf, select **Processing ⇨Start Simulation**. The simulation will then run with the new test vectors. If you modify Tcontrol.vhd, you will need to recompile first.

8.11 Running the Video Train System (After Successful Simulation)

A simulated or "virtual" train system is provided to test the controller without putting trains and people at risk. The simulation runs on the FPGA chip. The output of the simulation is displayed on a VGA monitor connected directly to the FPGA board. A typical video output display is seen in Figure 8.11. This module is also written in VHDL and it provides the sensor inputs and uses the outputs from the state machine to control the trains. The module tcontrol.vhd is automatically connected to the train simulation.

Here are the steps to run the virtual train system simulation:

Select the top-level project

Make Train.vhd the current project with **File ⇨Open Project ⇨*Name***

Then find and select **Train.qpf**. Train.qsf must be in the project directory since it contains the FPGA chip pin assignment information needed for video outputs and switch inputs. Double check that your FPGA Device type is correct.

Compile the Project

Select **Processing ⇨Start Compilation**. Train.vhd will link in your tcontrol.vhd file if it is in the same directory, when compiled. This is a large program, so it will take a few seconds to compile.

Download the Video Train Simulation

Select **Tools ⇨Programmer**. When the programmer window opens click on the Program/Configure box if it is not already selected. In case of problems, see the FPGA board download tutorials in Chapter 1 for more details. The FPGA board must be turned on the power supply must be connected, and the Byteblaster* cable must be plugged into the PC. When everything is setup, the start button in the programming window should highlight. If the start button is not highlighted, try closing and reopening the programmer window. Under Hardware setup the Byteblaster should be selected. To download the board, click on the highlighted **start** button. Attach a VGA monitor to the FPGA board.

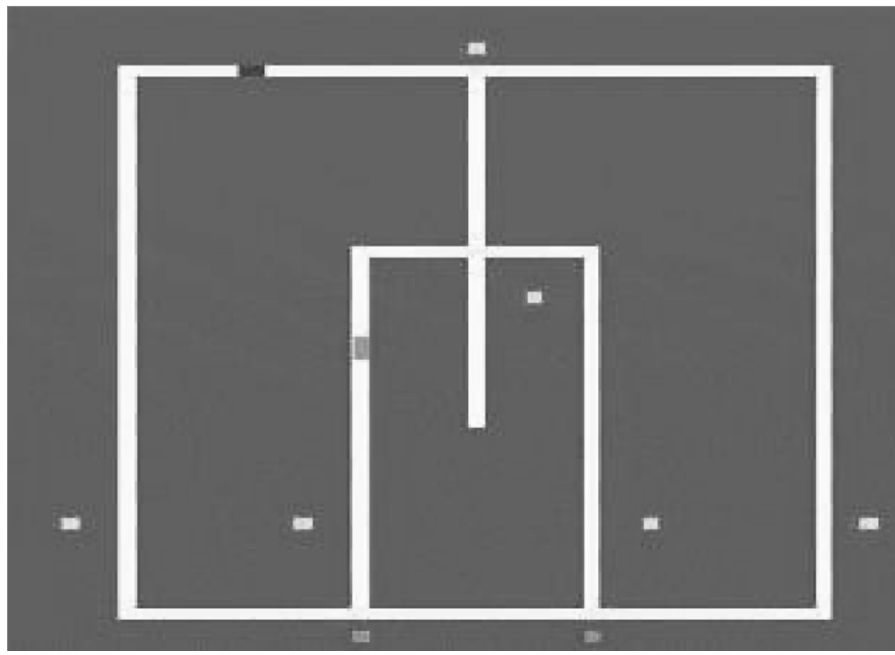


Figure 8.11 Video Image from Train System Simulation.

Viewing the Video Train Simulation

Train output should appear on the VGA monitor after downloading is complete as seen in Figure 8.11. On the DE1 and DE2, FPGA KEY2 is run/step/stop and FPGA KEY1 is the reset. Train A is displayed in black and Train B is displayed

in red. On the DE1 and DE2, hit KEY2 once to start the train simulation running. Hitting KEY2 again will stop the simulation. If you hit KEY2 twice quickly while trains are stopped, it will single step to the next track sensor state change. Other boards also use two pushbuttons for these functions.

Sensor and switch values are indicated with a green or red square on the display. Switch values are the squares next to each switch location. Green indicates no train present on a sensor and it indicates switch connected to outside track for a switch.

On the DE2 and UP3 board's, the LCD display top line shows the values of the sensor (s), and switch (sw) signals in binary and the bottom line indicates the values of DirA and DirB in binary. The most significant bit in each field is the highest numbered bit.

If a possible train wreck is detected by two trains running on the same track segment, the simulation halts and the monitor will flash. The FPGA board's slide or DIP switches control the speed of Train A (low 2 bits) and B (high 2 bits). Be sure to check operation with different train speeds. Many problems occur more often with a fast moving and a slow moving train.

8.12 A Hardware Implementation of the Train System Layout

Using the new Digital Command Control (DCC) model trains[3], a model train system with a similar track layout and control scheme can be setup and controlled by the FPGA board[4]. In DCC model trains, the train speed, direction, and other special features are controlled via a bipolar bit stream that is transmitted on the train tracks along with the power. A DCC decoder is located inside each train's engine that interprets the DCC signals and initiates the desired action (i.e., change in speed, direction, or feature status).

On a DCC system, trains are individually addressable. As seen in Figure 8.12, the DCC signal's frequency or zero crossing rate is changed in the DCC signal to transmit the data bits used for a command. A DCC command contains both a train address and a speed command. Each train engine is assigned a unique address. The electric motors in the train's engine are powered by a simple diode circuit that provides full–wave rectification of the bipolar DCC signal that is present on the track. In this way, the two metal train tracks can simultaneously provide both direct current (DC) power and speed control commands for the trains.

The output voltages and current levels provided by an FPGA output pin cannot drive the DCC train signals directly, but an FPGA can send the DCC data streams to the train track with the addition of a higher current H–bridge circuit that controls the train's power supply. An H-bridge contains four large power transistors that provide the higher drive current needed for DC motors and they

[3] DCC standards are approved by the National Model Railroad Association and are available online at http://www.nmra.org/standards/DCC/standards_rps/DCCStds.html.

[4] Additional details on using FPGAs for DCC can be found in "Using the Using FPGAs to Simulate and Implement Digital Design Systems in the Classroom", by T. S. Hall and J. O. Hamblen in the *Proceedings of the 2006 ASEE Southeast Section Conference*.

can also reverse the motor. Integrated H–bridge modules are available that can minimize the number of discrete components used. One such example is the National Semiconductor LMD18200 integrated H–bridge module. The LMD18200 supports TTL and CMOS compatible inputs allowing the FPGA board's output pins to be connected directly to the H-bridge inputs. A H-bridge typically requires two digital input control pins (i.e., forward, reverse, and stop). The H-bridge switches the train's power supply and in addition to the FPGA output pins that drive the H-bridge inputs, a ground connection is required between the train's power supply and the FPGA power supply.

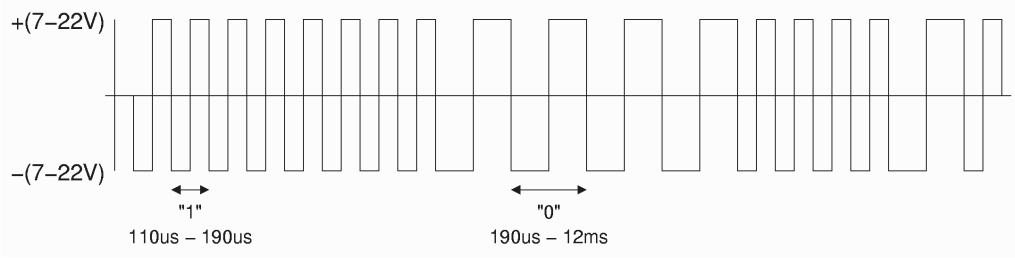


Figure 8.12 A portion of a DCC train signal is seen above. The zero crossing rate of a DCC signal is used to send data bits for train speed commands. The DCC signal is also rectified in each train's engine to provide 7-22V DC power for the train's electric motor and decoder circuits.

For the train sensors, Sharp GP2L26 infrared (IR) photointerrupter sensors can be used to detect when a train passes each sensor point. These sensors emit IR light from an LED and detect when the light is reflected back with an IR detector circuit. These sensors are very small (3mm x 4mm) and can fit between the rails on the track. Wires can be run down through the roadbed to a central protoboard where the discrete components needed for interfacing to this sensor are connected. Many model trains have dark underbodies, and the IR photo sensors can not always detect the trains passing over them. To increase the visibility of the trains to the photo sensors, pieces of reflective tape can be taped to the bottom of the trains.

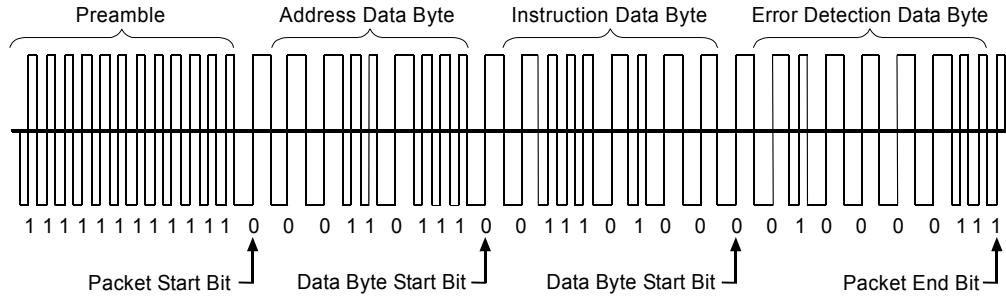


Figure 8.13 An example DCC model train speed and direction command packet.

Another version of the train.VHD program, dcc_train.zip is available on the book's DVD that will control a DCC train setup. It replaces the video train simulation module and produces the outputs needed to run the real DCC train system. It contains a DCC IP core, DCC_low_level.vhd, which generates the appropriate DCC signal packets as seen in Figure 8.13. The DCC data stream is generated by combining the Speed switch inputs and the Train direction signals (i.e., DA, DB) from the Tcontrol module. The appropriate DCC command packet is created from these signals and then saved in a register. The registered command is shifted out to produce a serial bit stream. The DCC standard only provides for one–way communication, and thus, no transmission guarantee can be made (i.e., no acknowledgement is sent back by the train). Therefore, a given DCC command is repeatedly shifted out until another command is received to ensure transmission of each command. Continuous transmission also insures a consistent power level on the tracks.

Additional construction details for anyone building the FPGA-based DCC model train setup are available at the book's website. The FPGA uses five input pins to read in from the IR photointerrupter track sensors, two output bits to send DCC commands, and two output bits to control each of the three track switches. Each track switch has two solenoid drivers that open and close a switch. The proper solenoid must be briefly turned on or pulsed to move the switch and then turned off. Leaving the solenoid turned on continuously will overheat and eventually burn out the solenoid. The 50ms timed pulse required to briefly energize a track switch's solenoid is already provided in the IP core.

To connect the train setup to all of the FPGA I/O pins, a ribbon cable can be attached to one of the I/O expansion headers on the FPGA board with the other end attached to the train interface circuitry on a protoboard or custom printed circuit board (PCB). The FPGA device type and I/O pin assignments for the train.VHD project will need to be changed depending on each user's custom train interface circuitry and the FPGA board I/O expansion connector used. Consult each FPGA board's reference manual for complete details on the I/O expansion header's FPGA pin numbers.

8.13 Laboratory Exercises

1. Assuming that train A now runs clockwise and B remains counterclockwise, draw a new state diagram and implement the new controller. If you use VHDL to design the new controller, you can modify the code presented in section 8.7. Simulate the controller and then run the video train simulation.

2. Design a state machine to operate the two trains avoiding collisions but minimizing their idle time. Trains must not crash by moving the wrong direction into an open switch. Develop a simulation to verify your state machine is operating correctly before running the video train system. The trains are assumed to be in the initial positions as shown in Figure 8.14. Train A is to move counterclockwise around the outside track until it comes to Sensor 1, then move to the inside track stopping at Sensor 5 and waiting for B to pass Sensor 3 twice. Trains can move at different speeds, so no assumption should be made

about the train speeds. A train hitting a sensor can be stopped before entering the switch area.

Once B has passed Sensor 3 twice, Train A moves to the outside track and continues around counterclockwise until it picks up where it left off at the starting position as shown in Figure 8.15. Train B is to move as designated only stopping at a sensor to avoid collisions with A. Train B will then continue as soon as there is no potential collision and continue as designated. Trains A and B should run continuously, stopping only to avoid a potential collision.

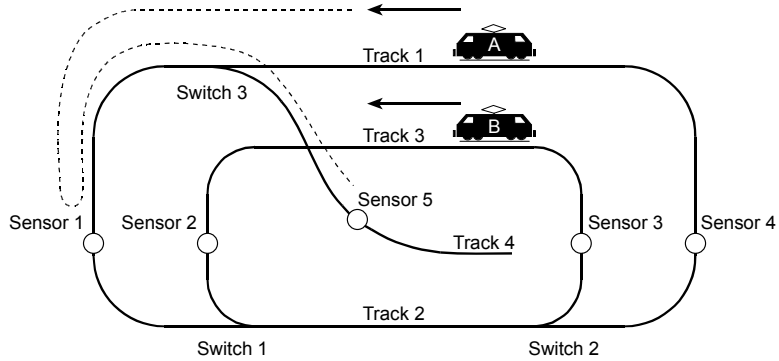


Figure 8.14 Initial Positions of Trains at State Machine Reset with Initial Paths Designated.

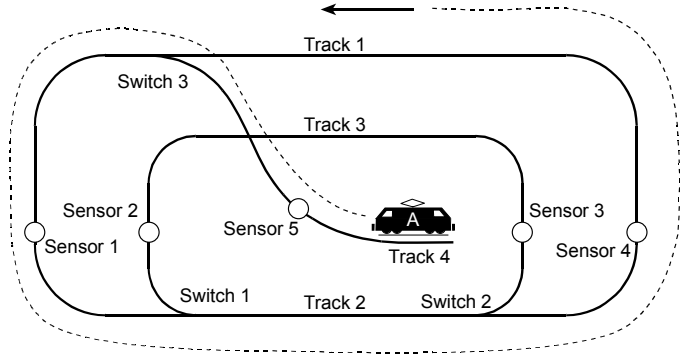


Figure 8.15 Return Path of Train A.

3. Use the single pulse FPGAcore functions on each raw sensor input to produce state machine sensor inputs that go High for only one clock cycle per passage of a train. Rework the state machine design with this assumption and repeat problem 1 or 2.
4. Develop another pattern of train movement and design a state machine to implement it.
5. Implement a real train setup using DCC model trains. Debug your control module using the video simulation module first, to avoid any real train crashes that may damage the trains. Typically laboratory space is limited, so keep in mind that the smaller gauge model trains will require less space for the track layout.