Tyler Kim

Digital Logic Design

Instruction Sequencer

Description

The purpose of this learning activity is to create an instruction sequencer for the cpu. The instruction sequencer will take in three inputs: *opcode(2..0)*, *reset*, and *clock* and will have eight outputs: *t0-t7* and *execute*. This component can be represented as a finite state machine (FSM) with eight states ranging from state 0 to state 7. At each state, the component will return a particular output.

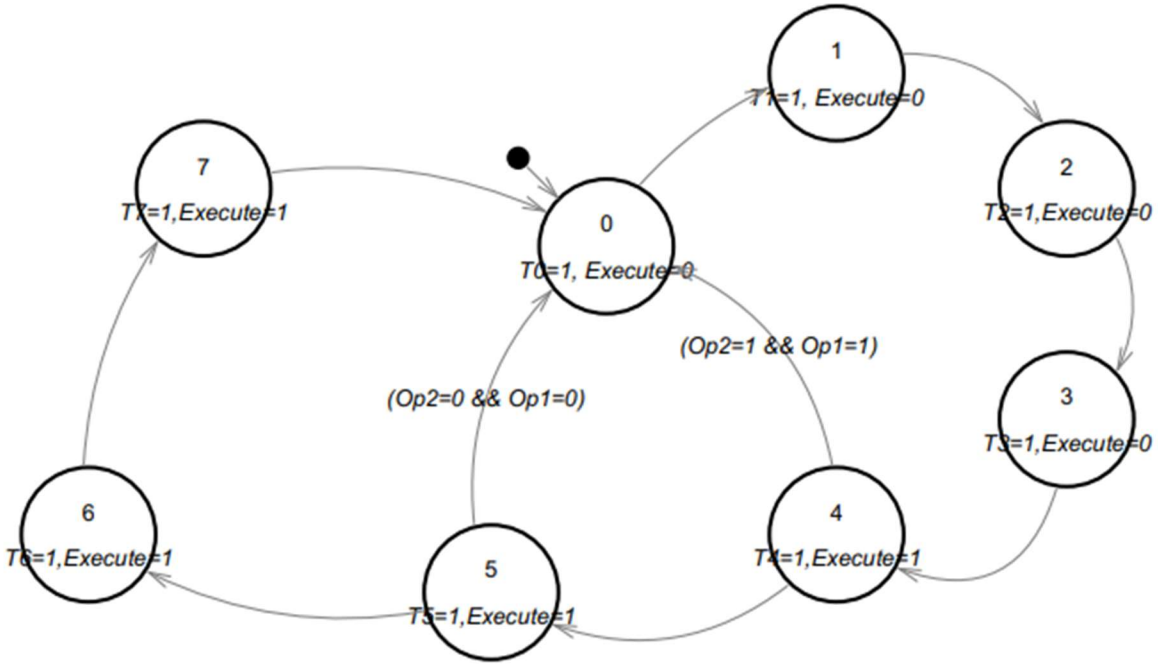


Figure 1: Finite State Machine of Instruction Sequencer (from Document)

Figure 1 represents the FSM for the instruction sequencer. In detail, the machine will start at state 0 and move to states 1 – 4 immediately. If *opcode(2)* is 1 and *opcode(1)* is 1, then the FSM will go back to state 0; if not, then it will continue and the same logic applies at state 5. The FSM can be represented in multiple tables.

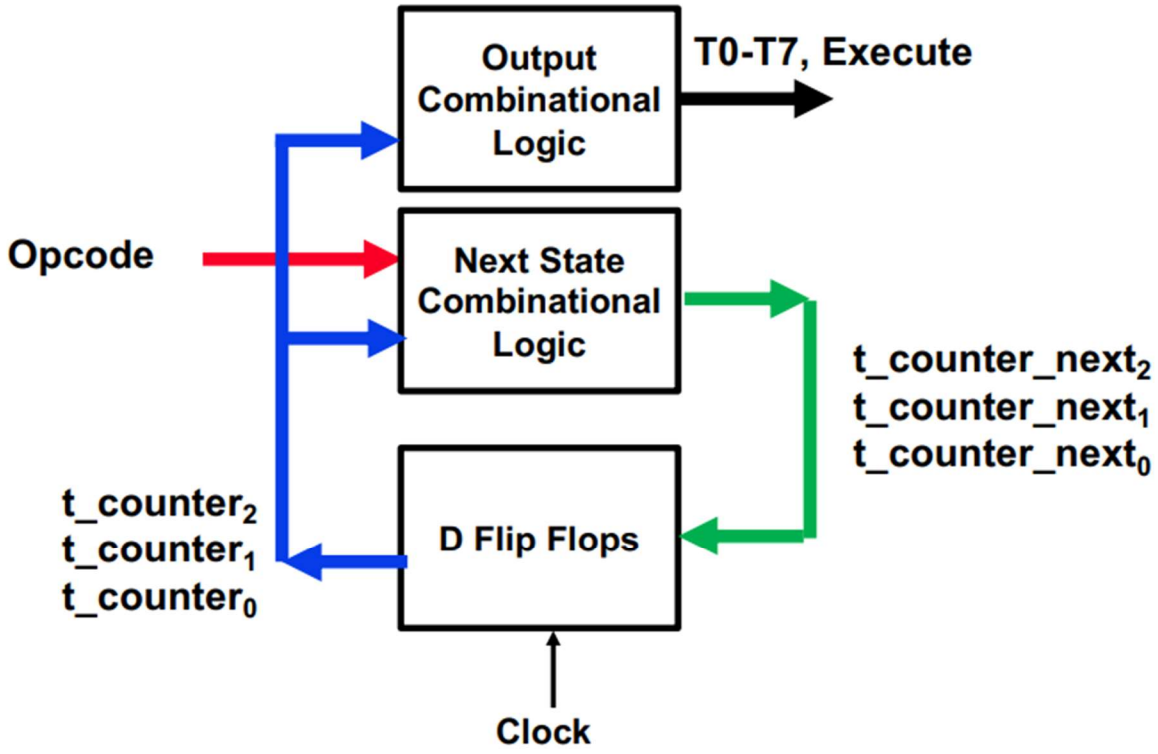


Figure 2: Overview of the Instruction Sequencer (from Slides)

Figure 2 highlights a flow diagram of the instruction sequencer. In detail, the *opcode* will be fed into to calculate the current and next states of the FSM like a Mealy Machine. The instruction sequencer will store the current and next states and output a signal to *t0-t7* and *execute*.

Design and Schematic

The design for the instruction sequencer requires multiple tables for the following: outputs *t0-t7*, next states, and output *execute*.

	opcode(2)	opcode(1)	t_counter(2)	t_counter(1)	t_counter(0)	t_counter_next(2)	t_counter_next(1)	t_counter_next(0)
0	0	0	0	0	0	0	0	1
1	0	0	0	0	1	0	1	0
2	0	0	0	1	0	0	1	1
3	0	0	0	1	1	1	0	0
4	0	0	1	0	0	1	0	1
5	0	0	1	0	1	0	0	0
6	0	0	1	1	0	x	x	x
7	0	0	1	1	1	x	x	x
8	0	1	0	0	0	0	0	1
9	0	1	0	0	1	0	1	0
10	0	1	0	1	0	0	1	1
11	0	1	0	1	1	1	0	0
12	0	1	1	0	0	1	0	1
13	0	1	1	0	1	1	1	0
14	0	1	1	1	0	1	1	1
15	0	1	1	1	1	0	0	0
16	1	0	0	0	0	0	0	1
17	1	0	0	0	1	0	1	0
18	1	0	0	1	0	0	1	1
19	1	0	0	1	1	1	0	0
20	1	0	1	0	0	1	0	1
21	1	0	1	0	1	1	1	0
22	1	0	1	1	0	1	1	1
23	1	0	1	1	1	0	0	0
24	1	1	0	0	0	0	0	1
25	1	1	0	0	1	0	1	0
26	1	1	0	1	0	0	1	1
27	1	1	0	1	1	1	0	0
28	1	1	1	0	0	0	0	0
29	1	1	1	0	1	x	x	x
30	1	1	1	1	0	x	x	x
31	1	1	1	1	1	x	x	x

Figure 3: Next State Table (from Slides)

Inputs: Opcode Current State	Output: T0-T7 Load (000)	Store (001)	Add (010)	Sub (011)	Inc (100)	Dec (101)	Bra (110)	Beq (111)
000	T0	T0	T0	T0	T0	T0	T0	T0
001	T1	T1	T1	T1	T1	T1	T1	T1
010	T2	T2	T2	T2	T2	T2	T2	T2
011	T3	T3	T3	T3	T3	T3	T3	T3
100	T4	T4	T4	T4	T4	T4	T4	T4
101	T5	T5	T5	T5	T5	T5	x	x
110	x	x	T6	T6	T6	T6	x	x
111	x	x	T7	T7	T7	T7	x	x

Figure 4: T0-T7 Output Table (from Document)

Inputs: Opcode / Current State	Output: Execute Load (000)	Store (001)	Add (010)	Sub (011)	Inc (100)	Dec (101)	Bra (110)	Beq (111)
000	0	0	0	0	0	0	0	0
001	0	0	0	0	0	0	0	0
010	0	0	0	0	0	0	0	0
011	0	0	0	0	0	0	0	0
100	1	1	1	1	1	1	1	1
101	1	1	1	1	1	1	x	x
110	x	x	1	1	1	1	x	x
111	x	x	1	1	1	1	x	x

Figure 5: Execute Output Table (from Document)

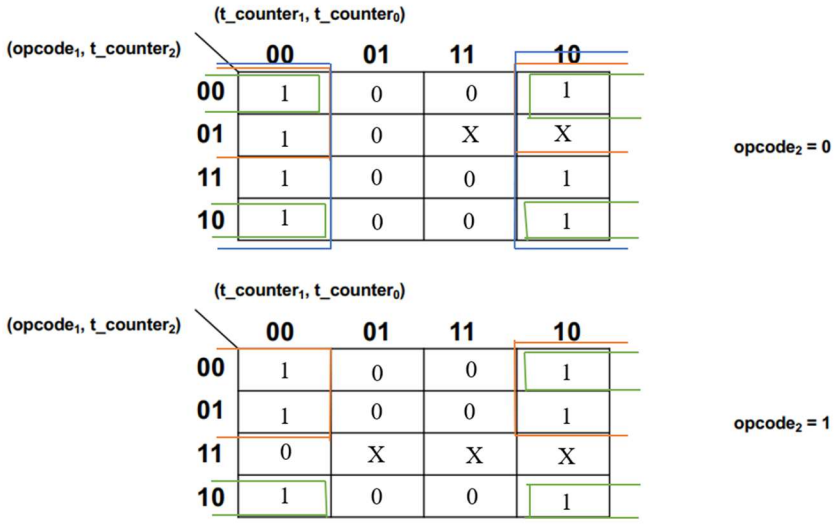
Developing the schematic required three steps: the current state logic, the next state logic, and the output logic.

Current State Logic

Designing the current state logic only required d-flipflops. Since the Mealy machine required 8 states, three state variables were used to represent all eight states. D-flipflops were used to store each state variable where the output *q* represents the current state and the input *d* stores the next state. Every time a state transition occurs, the flipflop will update the current state to the next state; thus, the cycle continues. In other words, current state logic only required storing the current state as *q* in the d-flipflops.

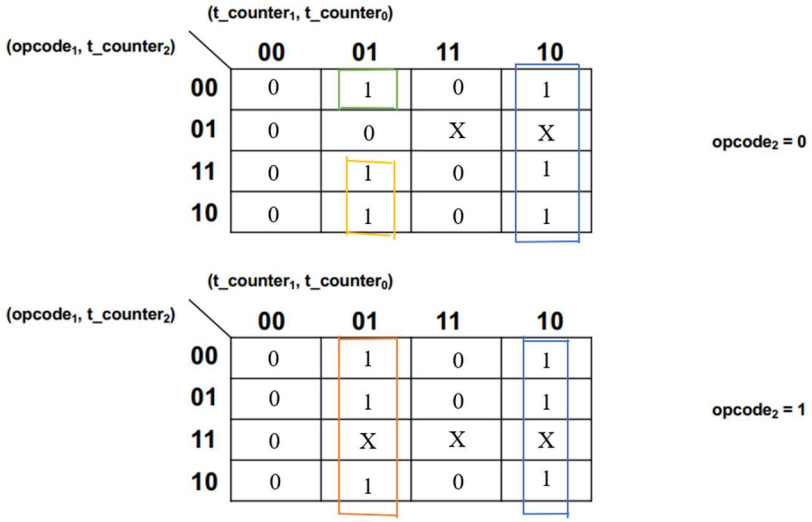
Next State Logic

The next state logic required Karnaugh maps in order to develop the schematic for each of the three next state variables. Figure 3 represents the next state table and was thus used to develop the Karnaugh maps.



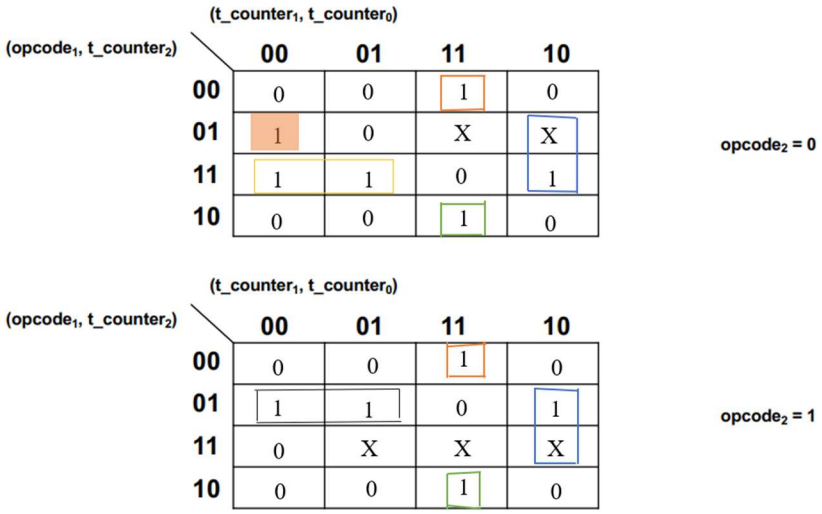
t_counter_next[0] = (opcode[1]' and t_counter[0]') or (opcode[2]' and t_counter[0]') or (t_counter[2]' and t_counter[0]')

Figure 6: Karnaugh Map for t_counter_next[0]



t_counter_next[0] = (t_counter[0] and t_counter[0]') or (opcode(2) and t_counter[1]' and t_counter[0]) or (opcode(2)' and opcode(1)' and t_counter[2]' and t_counter[1]' and t_counter[0]) or (opcode[2]' and opcode[1] and t_counter[1]' and t_counter[0])

Figure 7: Karnaugh Map for t_counter_next[1]



t_counter_next[0] = (t_counter[2] and t_counter[1] and t_counter[0]') or (opcode[1]' and t_counter[2]' and t_counter[1] and t_counter[0]) or (opcode[1] and t_counter[2] and t_counter[1] and t_counter[0]) or (opcode[2] and opcode[1]' and t_counter[2] and t_counter[1]') or (opcode[2]' and opcode[1] and t_counter[2] and t_counter[1]') or (opcode[2]' and t_counter[2] and t_counter[1]' and t_counter[0]')

Figure 8: Karnaugh Map for t_counter_next[2]

Figures 6-8 show the k-maps for each of the next state variables using the table from Figure 3. The d-flipflops input *d* were assigned the values of their respective next state variables so that they can be loaded when transitioning to the next state. The next state logic itself was created with combinational circuits.

Output Logic

The final step to create the output logic was primarily combinational logic. The *t0-t7* output was created using Figure 4. A K-map was not needed for designing the output logic for *t0-t7* because the outputs were associated with the combination of the state variables. In other words, if the state variables were 010, then the FSM would output *t2* as 1. Simple AND and OR gates were used to create the outputs from *t0-t7*.

In the case for the *execute*, a K-map was not needed because *execute* yield 1 of the combination of the state variables yielded a decimal value greater than or equal to 4. In other words, *execute*

was equal to the most significant bit *t_counter[2]* or *q[2]* in the case of my particular implementation of the instruction sequencer.

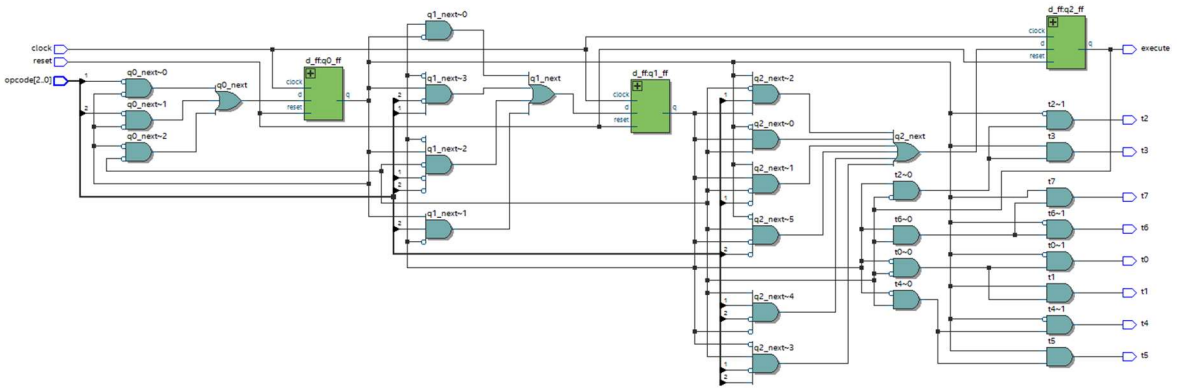


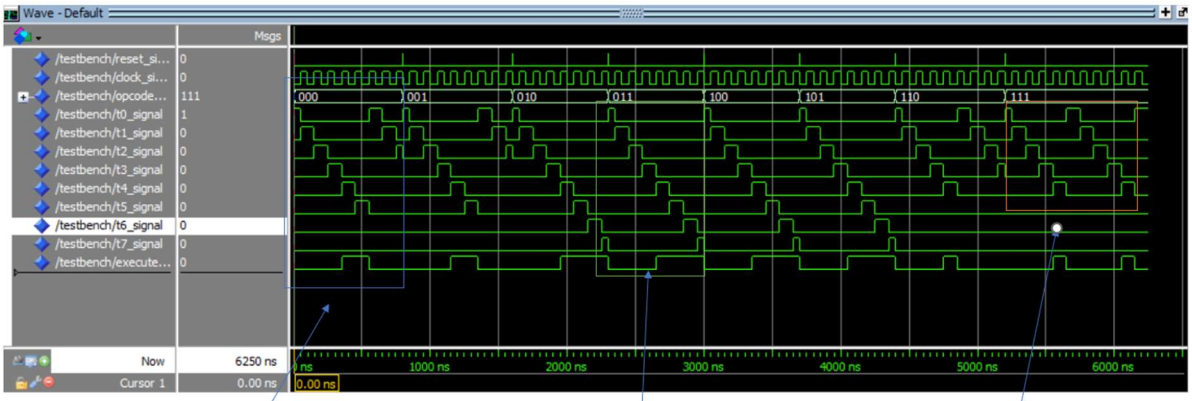
Figure 9: Schematic for Instruction Sequencer

Verification

Inputs: Opcode / Current State	Next State: Load (000)	Store (001)	Add (010)	Sub (011)	Inc (100)	Dec (101)	Bra (110)	Beq (111)
000	001	001	001	001	001	001	001	001
001	010	010	010	010	010	010	010	010
010	011	011	011	011	011	011	011	011
011	100	100	100	100	100	100	100	100
100	101	101	101	101	101	101	000	000
101	000	000	110	110	110	110	x	x
110	x	x	111	111	111	111	x	x
111	x	x	000	000	000	000	x	x

Figure 10: Next State Transition Table (from Document)

Figure 9 highlights the schematic for the instruction sequencer. Verification for the instruction sequencer was done in two ways: signal waves and assertions. The results of the signal waves were compared with Figure 10 and the assertion provide a quicker way to evaluate the correctness of the schematic.



For when opcode = 000, the signal wave illustrates a "staircase" pattern indicating that the instruction sequencer is going through all the steps and appropriately outputs the execute signal. In addition, the signal wave indicates that it immediately goes back to t0.

Similar case except when opcode = 011, more microinstructions are called making execute = 1 for a long period of time.

For opcode = 111, fewer microinstructions are called so execute = 1 for a short period of time

Figure 10: Annotated Verification

Inputs: Opcode / Current State	Next State Load (000)	Store (001)	Add (010)	Sub (011)	Inc (100)	Dec (101)	Bra (110)	Beq (111)
000	001	001	001	001	001	001	001	001
001	010	010	010	010	010	010	010	010
010	011	011	011	011	011	011	011	011
011	100	100	100	100	100	100	100	100
100	101	101	101	101	101	101	000	000
101	000	000	110	110	110	110	x	x
110	x	x	111	111	111	111	x	x
111	x	x	000	000	000	000	x	x

Figure 11: Verification with Signal Waves and Comparison with Table

```
# ** Note: T0 correct for  opcode = 000
#    Time: 3 ns  Iteration: 0  Instance: /testbench
# ** Note: T1 correct for  opcode = 000
#    Time: 100 ns  Iteration: 1  Instance: /testbench
# ** Note: T2 correct for  opcode = 000
#    Time: 200 ns  Iteration: 1  Instance: /testbench
# ** Note: T3 correct for  opcode = 000
#    Time: 300 ns  Iteration: 1  Instance: /testbench
# ** Note: T4 correct for  opcode = 000
#    Time: 400 ns  Iteration: 1  Instance: /testbench
# ** Note: T5 correct for  opcode = 000
#    Time: 500 ns  Iteration: 1  Instance: /testbench
# ** Note: T0 correct for  opcode = 001
#    Time: 503 ns  Iteration: 0  Instance: /testbench
# ** Note: T1 correct for  opcode = 001
#    Time: 600 ns  Iteration: 1  Instance: /testbench
# ** Note: T2 correct for  opcode = 001
#    Time: 700 ns  Iteration: 1  Instance: /testbench
# ** Note: T3 correct for  opcode = 001
#    Time: 800 ns  Iteration: 1  Instance: /testbench
# ** Note: T4 correct for  opcode = 001
#    Time: 900 ns  Iteration: 1  Instance: /testbench
# ** Note: T5 correct for  opcode = 001
#    Time: 1 us  Iteration: 1  Instance: /testbench
# ** Note: T0 correct for  opcode = 010
#    Time: 1003 ns  Iteration: 0  Instance: /testbench
# ** Note: T1 correct for  opcode = 010
#    Time: 1100 ns  Iteration: 1  Instance: /testbench
# ** Note: T2 correct for  opcode = 010
#    Time: 1200 ns  Iteration: 1  Instance: /testbench
# ** Note: T3 correct for  opcode = 010
#    Time: 1300 ns  Iteration: 1  Instance: /testbench
# ** Note: T4 correct for  opcode = 010
#    Time: 1400 ns  Iteration: 1  Instance: /testbench
# ** Note: T5 correct for  opcode = 010
#    Time: 1500 ns  Iteration: 1  Instance: /testbench
# ** Note: T6 correct for  opcode = 010
#    Time: 1600 ns  Iteration: 1  Instance: /testbench
# ** Note: T7 correct for  opcode = 010
#    Time: 1700 ns  Iteration: 1  Instance: /testbench
# ** Note: T0 correct for  opcode = 011
#    Time: 1703 ns  Iteration: 0  Instance: /testbench
# ** Note: T1 correct for  opcode = 011
#    Time: 1800 ns  Iteration: 1  Instance: /testbench
# ** Note: T2 correct for  opcode = 011
#    Time: 1900 ns  Iteration: 1  Instance: /testbench
# ** Note: T3 correct for  opcode = 011
#    Time: 2 us  Iteration: 1  Instance: /testbench
# ** Note: T4 correct for  opcode = 011
#    Time: 2100 ns  Iteration: 1  Instance: /testbench
# ** Note: T5 correct for  opcode = 011
#    Time: 2200 ns  Iteration: 1  Instance: /testbench
# ** Note: T6 correct for  opcode = 011
#    Time: 2300 ns  Iteration: 1  Instance: /testbench
# ** Note: T7 correct for  opcode = 011
```

Figure 12: Verification for Opcodes 000 – 011

```
# ** Note: T0 correct for  opcode = 100
#    Time: 2403 ns  Iteration: 0  Instance: /testbench
# ** Note: T1 correct for  opcode = 100
#    Time: 2500 ns  Iteration: 1  Instance: /testbench
# ** Note: T2 correct for  opcode = 100
#    Time: 2600 ns  Iteration: 1  Instance: /testbench
# ** Note: T3 correct for  opcode = 100
#    Time: 2700 ns  Iteration: 1  Instance: /testbench
# ** Note: T4 correct for  opcode = 100
#    Time: 2800 ns  Iteration: 1  Instance: /testbench
# ** Note: T5 correct for  opcode = 100
#    Time: 2900 ns  Iteration: 1  Instance: /testbench
# ** Note: T5 correct for  opcode = 100
#    Time: 3 us  Iteration: 1  Instance: /testbench
# ** Note: T6 correct for  opcode = 100
#    Time: 3100 ns  Iteration: 1  Instance: /testbench
# ** Note: T0 correct for  opcode = 101
#    Time: 3103 ns  Iteration: 0  Instance: /testbench
# ** Note: T1 correct for  opcode = 101
#    Time: 3200 ns  Iteration: 1  Instance: /testbench
# ** Note: T2 correct for  opcode = 101
#    Time: 3300 ns  Iteration: 1  Instance: /testbench
# ** Note: T3 correct for  opcode = 101
#    Time: 3400 ns  Iteration: 1  Instance: /testbench
# ** Note: T4 correct for  opcode = 101
#    Time: 3500 ns  Iteration: 1  Instance: /testbench
# ** Note: T5 correct for  opcode = 101
#    Time: 3600 ns  Iteration: 1  Instance: /testbench
# ** Note: T6 correct for  opcode = 101
#    Time: 3700 ns  Iteration: 1  Instance: /testbench
# ** Note: T7 correct for  opcode = 101
#    Time: 3800 ns  Iteration: 1  Instance: /testbench
# ** Note: T0 correct for  opcode = 110
#    Time: 3803 ns  Iteration: 0  Instance: /testbench
# ** Note: T1 correct for  opcode = 110
#    Time: 3900 ns  Iteration: 1  Instance: /testbench
# ** Note: T2 correct for  opcode = 110
#    Time: 4 us  Iteration: 1  Instance: /testbench
# ** Note: T3 correct for  opcode = 110
#    Time: 4100 ns  Iteration: 1  Instance: /testbench
# ** Note: T4 correct for  opcode = 110
#    Time: 4200 ns  Iteration: 1  Instance: /testbench
# ** Note: T0 correct for  opcode = 111
#    Time: 4203 ns  Iteration: 0  Instance: /testbench
# ** Note: T1 correct for  opcode = 111
#    Time: 4300 ns  Iteration: 1  Instance: /testbench
# ** Note: T2 correct for  opcode = 111
#    Time: 4400 ns  Iteration: 1  Instance: /testbench
# ** Note: T3 correct for  opcode = 111
#    Time: 4500 ns  Iteration: 1  Instance: /testbench
# ** Note: T4 correct for  opcode = 111
#    Time: 4600 ns  Iteration: 1  Instance: /testbench
```

Figure 13: Verification for Opcodes 100 – 111

As evident from the assertion statements and verification, the instruction sequencer successfully fulfilled the constraints and requirements of the document. Therefore, the instruction sequencer was successfully designed.