APPENDIX C

# Extended RTL Design Example

## C.1 INTRODUCTION

Chapter 5 included an RTL design example of a soda dispenser processor. The example started with a high-level state machine, created the datapath's structure, and then described the controller using a finite-state machine. The controller was not further design to structure, as such design was the subject of Chapter 3. This appendix completes the RTL design by designing the controller's FSM down to a state register and gates, resulting in a complete custom-processor implementation of a controller and a datapath. The appendix traces through the behavior of the complete implementation. The purpose of demonstrating this complete design is to give the reader a clear understanding of how the controller and datapath work together.

The block symbol for the soda dispenser processor appears in Figure C.1. Recall that the soda dispenser has three inputs `c`, `s`, and `a`. The 8-bit input `s` represents the cost of each bottle of soda. The 1-bit input `c` is `1` for one clock cycle when a coin is inserted. Additionally, the value on 8-bit input `a` indicates the value of the coin that was inserted. The soda dispenser features one output `d` used to indicate when soda should be dispensed. The 1-bit output `d` is `1` for one clock cycle after the value of the coins inserted into the soda dispenser is greater than or equal to `s`. The soda dispenser does not give change.

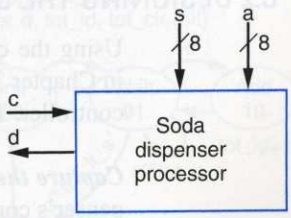



**Figure C.1** Soda dispenser block symbol.

Chapter 5 developed the high-level state machine seen in Figure C.2. The HLSM was then converted into a controller (represented behaviorally as an FSM) and datapath, shown in Figure C.3.

The datapath supports the data operations necessitated by the high-level state machine, including resetting the value of *tot* (*tot* = `0` in the *Init* state), comparing whether *tot* is less than `s` (for the transitions from the *Wait* state), and adding *tot* and `a` (in the *Add* state).

The controller FSM is similar to the high-level state machine, but is modified to control the datapath and accept status input from the datapath (i.e., `tot_lt_s`) rather than performing data operations directly. The controller and datapath are shown in Figure C.3.

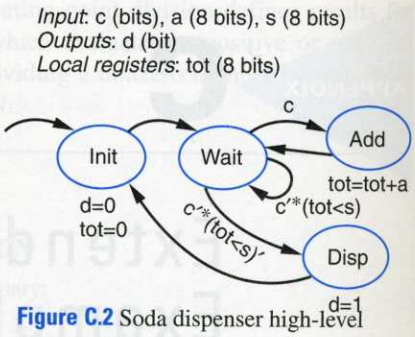



**Figure C.2** Soda dispenser high-level state machine.

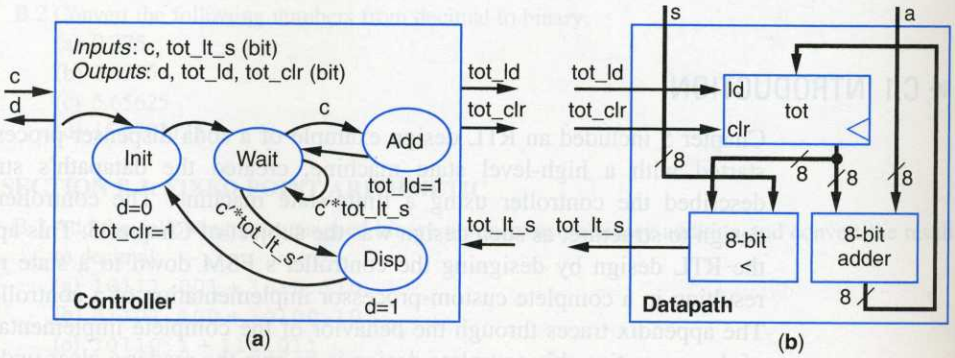



**Figure C.3** Soda dispenser: (a) controller (described behaviorally) and (b) datapath (structure).

## C.2 DESIGNING THE SODA DISPENSER CONTROLLER

Using the controller design process introduced in Chapter 3, we can complete the design of the controller. The steps are as follows:

***Capture the FSM.*** The FSM for the soda dispenser's controller was created during Step 4 of the RTL design method. The controller's FSM is shown in Figure C.3(a).

***Set up the architecture.*** As indicated by the controller's FSM, the state machine's architecture has 2 inputs (`c` and `tot_lt_s`) and 3 outputs (`d`, `tot_ld`, and `tot_clr`). We will use two bits to represent the controller's states, naming the current state signals `s1` and `s0`, and the next state signals `n1` and `n0`. The corresponding controller architecture is shown in Figure C.4.

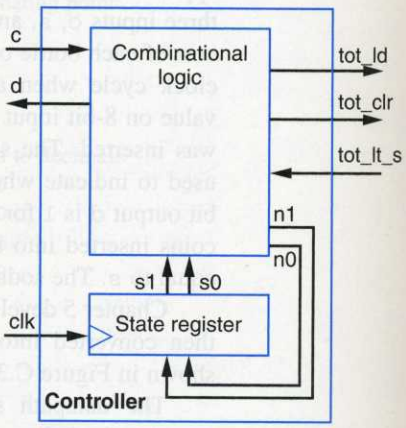



**Figure C.4** Standard controller architecture for the soda dispenser.

***Encode the states.*** A straightforward encoding of the soda dispenser's four states is: *Init*: `00`, *Wait*: `01`, *Add*: `10`, and *Disp*: `11`.

***Fill in the truth table.*** From the controller architecture set up above, we know that the truth table must have 4 inputs (`c`, `tot_lt_s`, `s1`, and `s0`) and 5 outputs (`d`, `tot_ld`, `tot_clr`, `n1`, and `n0`). With 4 inputs, the table will have $2^4 = 16$ rows as in Figure C.5.

| | Inputs | | | | Outputs | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | s1 | s0 | c | tot_lt_s | d | tot_ld | tot_clr | n1 | n0 |
| *Init* | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| *Wait* | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| *Add* | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| *Disp* | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |

**Figure C.5** The soda dispenser controller's truth table.

By examining the outputs specified in the controller FSM, duplicated for convenience in Figure C.6, we can fill in the `d`, `tot_ld`, and `tot_clr` columns in the truth table. For example, Figure C.6 shows that when the controller FSM is in state *Init*, `d=0`, `tot_clr=1`, and `tot_ld` is implicitly `0`. Thus, for rows in the truth table that correspond to state *Init* —namely the four rows where `s1s0=00`, since we chose "`00`" as the encoding for the *Init* state—we set the `d` column to `0`, the `tot_clr` column to `1`, and the `tot_ld` column to `0`.

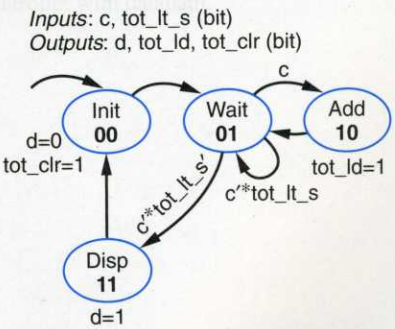



**Figure C.6** Soda dispenser controller FSM with state encodings.

We fill in the next state columns `n1` and `n0` based on the transitions specified in the controller FSM and the state encoding chosen in the earlier step. For example, consider the *Wait* state. As indicated in Figure C.6, the FSM transitions to the *Add* state when `c=1`. Thus, for rows where `s1s0c=011` (`s1s0=01` corresponds to the *Wait* state), we set the `n1` column to `1` and the `n0` column to `0` (`n1n0=10` corresponds to the *Add* state). When `c=0`, the FSM transitions to the *Disp* state if `tot_lt_s=0` or remains in the *Wait* state of `tot_lt_s=1`. We represent the transition from *Wait* to *Disp* in the truth table by setting

n1 to 1 and n0 to 1 (*Disp*) in the row where s1s0=01 (*Wait*), c=0, and tot_lt_s=0. Similarly, we represent the transition from *Wait* back to *Wait* by writing n1n0=01 where s1s0=01, c=0, and tot_lt_s=1. We then examine the remaining transitions in a similar way, filling in the appropriate values for n1 and n0 until all transitions are accounted for. The completed truth table is shown in Figure C.5.

***Implement the Combinational Logic.*** For each of the truth table's outputs, we write the corresponding Boolean equation. From the truth table we obtain the following equations:

```
d = s1s0
tot_ld = s1s0'
tot_clr = s1's0'
n1 = s1's0c'tot_lt_s' + s1's0c

n0 = s1's0' + s1's0c' + s1s0'
n0 = s0' + s1's0c'
```

Note that the first four equations derived from the truth table are already minimized. The fifth equation, corresponding to n0, can be minimized to s0' + s1's0c' through algebraic methods, or by using a K-map as shown in Figure C.7. K-maps were discussed in Chapter 6.

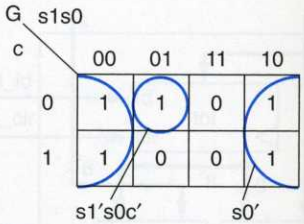



**Figure C.7** K-map for output n0.

Using techniques discussed in Chapter 2, we convert the above Boolean equations into an equivalent two-level gate-based circuit. This conversion is straightforward since the Boolean equations are already in sum-of-products form. The final sequential controller circuit and the datapath for the soda dispenser is shown in Figure C.8.

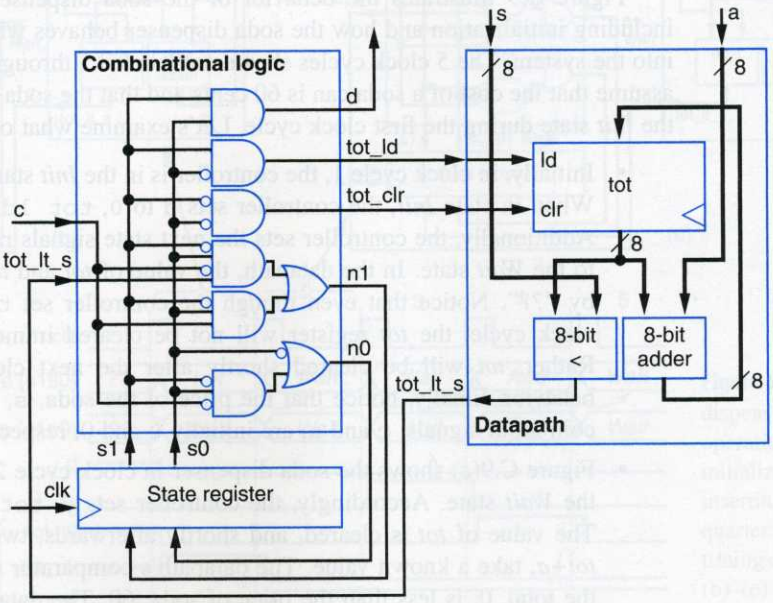


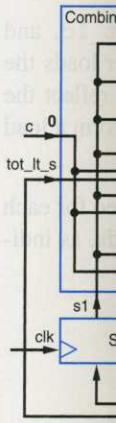

**Figure C.8** Final implementation of the soda machine controller with datapath.

## C.3 UNDERSTANDING THE BEHAVIOR OF THE SODA DISPENSER CONTROLLER AND DATAPATH

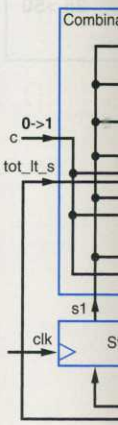
This section examines how the controller and datapath designed for the soda dispenser interact to form a working implementation of the initial high-level state machine.

Figure C.9 illustrates the behavior of the soda dispenser controller and datapath, including initialization and how the soda dispenser behaves when a user inserts a quarter into the system. The 5 clock cycles shown are labeled 1 through 5 in Figure C.9(a). We'll assume that the cost of a soda can is 60 cents and that the soda dispenser's controller is in the *Init* state during the first clock cycle. Let's examine what occurs in each clock cycle:

- Initially, in clock cycle 1, the controller is in the *Init* state, shown in Figure C.9(b). When in state *Init*, the controller sets `d` to `0`, `tot_ld` to `0`, and `tot_clr` to `1`. Additionally, the controller sets the next state signals `n1n0` to `01`, corresponding to the *Wait* state. In the datapath, the value of *tot* and *tot+a* is unknown, denoted by "*??*". Notice that even though the controller set `tot_clr` to `1` during this clock cycle, the *tot* register will not be cleared immediately (asynchronously). Rather, *tot* will be cleared shortly after the next clock cycle, a synchronous behavior. Finally, notice that the price of the soda, `s`, is set to 60 cents and the coin input signals, `c` and `a`, are initially `0` and 0, respectively.
- Figure C.9(c) shows the soda dispenser in clock cycle 2. The controller is now in the *Wait* state. Accordingly, the controller sets `d`, `tot_ld`, and `tot_clr` to `0`. The value of *tot* is cleared, and shortly afterwards, two signals, `tot_lt_s` and *tot+a*, take a known value. The datapath's comparator sets `tot_lt_s` to `1` since the total, 0, is less than the price of soda, 60. The datapath's adder sets intermediate signal *tot+a* to 0 since *tot* and `a` are now known. The next state signals remain set to `01` (*Wait*) since `c` is `0` and `tot_lt_s` is `1`.
- Figure C.9(d) shows the soda dispenser in clock cycle 3. During the third clock cycle, the user inserts a quarter into the soda dispenser, as indicated by `c` becoming `1` and `a` becoming 25. Shortly after `a` changes, the adder's output *tot+a* changes to 25, the sum of *tot* and `a`. Since `c` is `1`, the controller sets the next state to `10` (*Add*). The values of `d`, `tot_ld`, and `tot_clr` remain the same since the controller's state has not changed since the previous (2nd) clock cycle.
- In clock cycle 4, shown in Figure C.9(e), the controller is in the *Add* state and sets `tot_ld` to `1` while keeping `d` and `tot_clr` at `0`. As was the case with `tot_clr` during the *Init* state, *tot* will not be updated until the next clock cycle. The controller will unconditionally return to state *Wait*, setting `n1n0` to `01` (*Wait*).




(a)



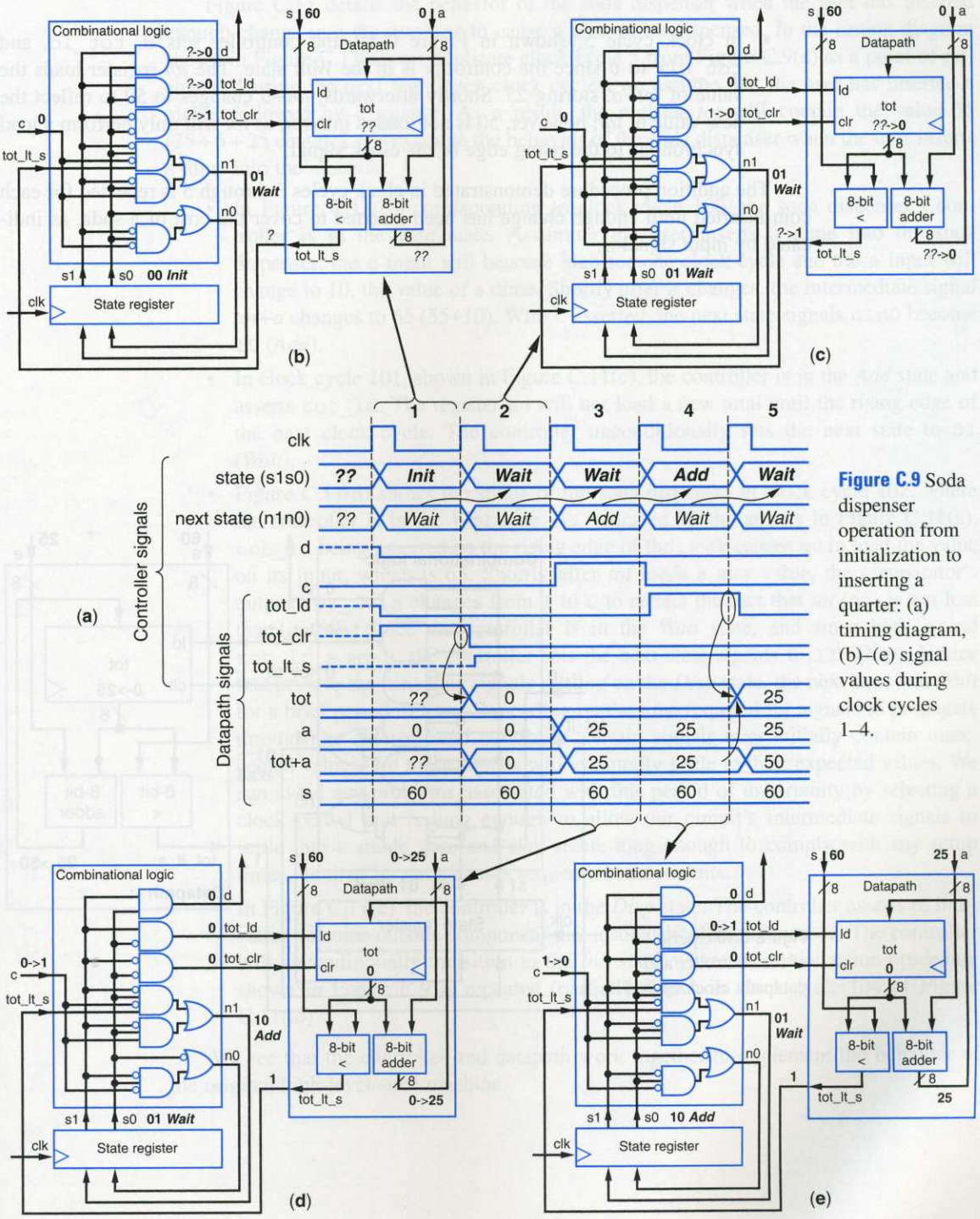



**Figure C.9** Soda dispenser operation from initialization to inserting a quarter: (a) timing diagram, (b)–(e) signal values during clock cycles 1–4.

- In clock cycle 5, shown in Figure C.10, the controller sets `d`, `tot_ld`, and `tot_clr` to `0` since the controller is in the *Wait* state. The *tot* register loads the value of *tot+a*, storing 25. Shortly afterwards, *tot+a* changes to 50 to reflect the new value of *tot*; however, 50 is not loaded into *tot*, as *tot* will only perform a load synchronous to the rising edge of the clock signal.

The addition procedure demonstrated in clock cycles 3 through 5 is repeated for each coin inserted until enough change has been inserted to cover the cost of a soda, as indicated by input signal `s`.

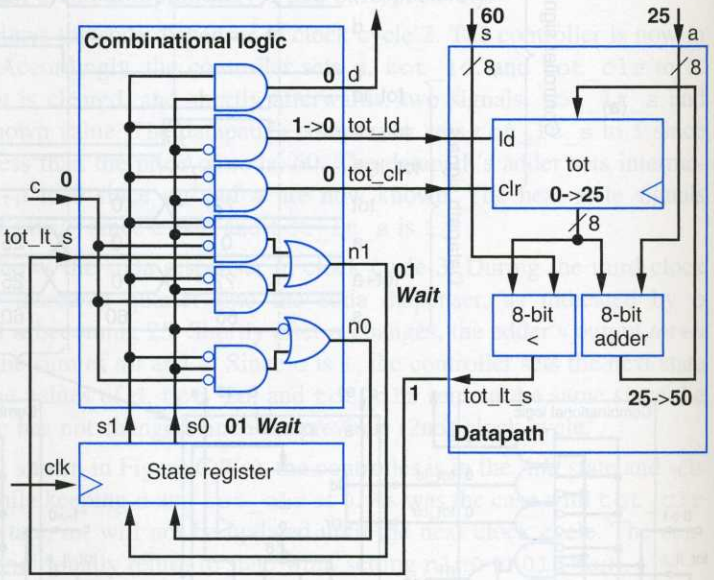



**Figure C.10** Operation of the controller and datapath: clock cycle 5.

Figure C.11 details the behavior of the soda dispenser when the user has inserted enough change into the machine to cause a soda to be dispensed. In the timing diagram shown in Figure C.11(a), we duplicate clock cycle 5 from Figure C.9(a) as a point of reference. During the next few dozen clock cycles, we assume that the user has inserted a nickel followed by a quarter. As a result, the register *tot* will contain the value 55 (25 + 5 + 25 cents). Let's examine the behavior of the soda dispenser when the user inserts a dime into the machine:

- In Figure C.11(b), corresponding to clock cycle 100, the soda dispenser's controller is in the *Wait* state. Assuming the user inserts a dime into the soda dispenser, the `c` input will become high for one clock cycle and the `a` input will change to 10, the value of a dime. Shortly after `a` changes, the intermediate signal *tot+a* changes to 65 (55+10). With `c` asserted, the next state signals `n1n0` become `10` (*Add*).
- In clock cycle 101, shown in Figure C.11(c), the controller is in the *Add* state and asserts `tot_ld`. The register *tot* will not load a new total until the rising edge of the next clock cycle. The controller unconditionally sets the next state to `01` (*Wait*).
- Figure C.11(d) shows the status of the soda dispenser in clock cycle 102, where the controller is in the *Wait* state. As indicated by the arrows in Figure C.11(a), `tot_ld` being asserted on the rising edge of the clock causes *tot* to load the value on its input, which is 65. Shortly after *tot* loads a new value, the comparator's output `tot_lt_s` changes from `1` to `0` to reflect the fact that *tot* (65) is not less than `s` (60). Since the controller is in the *Wait* state, and since both `c` and `tot_lt_s` are `0`, the controller sets the next state signals to `11` (*Disp*). Notice that prior to the next state signals settling on the *Disp* state, the next state was *Wait* for a brief period of time. Depending on the time required for signals to propagate through the datapath and controller, certain signals may initially contain unexpected values, but these signals will eventually settle to their expected values. We can avoid any problems associated with this period of uncertainty by selecting a clock period that is long enough to allow our circuit's intermediate signals to settle into a stable state and stay stable long enough to comply with any setup times required by our circuit's sequential components.
- In Figure C.11(e), the controller is in the *Disp* state. The controller asserts `d`, indicating to some outside component that a soda should be dispensed. The controller will unconditionally transition to the *Init* state, where the initialization procedure shown in Figure C.9 is repeated (partially shown in clock cycle 104 of Figure C.11(a)).

We see that the controller and datapath work together to implement the behavior of the original high-level state machine.

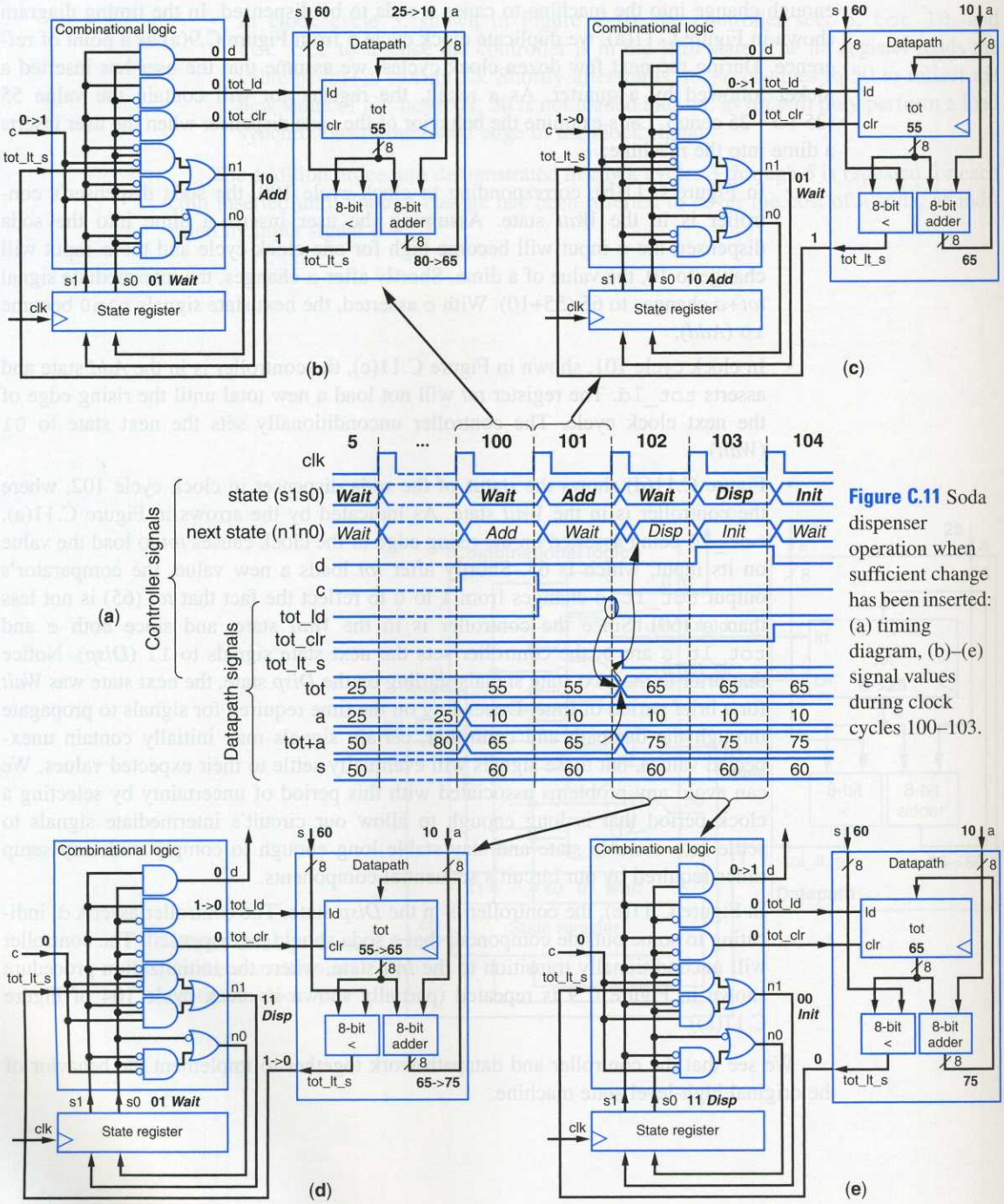



**Figure C.11** Soda dispenser operation when sufficient change has been inserted: (a) timing diagram, (b)–(e) signal values during clock cycles 100–103.