Use Abstraction to Design a Vending Machine FSM

Let’s build a more realistic vending machine. We’ll use several components:
• registers,
• adders,
• muxes, and
• decoders.
We’ll also develop a new component, priority encoders.
And one module specific to this FSM design.

Let’s Assume that Our Machine Sells Three Items

How many items should our vending machine sell?
Each item has
• a price,
• an input to identify it (such as a button), and
• an output to release it.
Three items makes the problem
• large enough to be interesting, but
• small enough to allow detailed illustration.

General Protocol for a Vending Machine

1. A user sees an item that they want to buy.
2. The user puts money into the machine.
3. The machine (FSM) keeps track of how much money has been inserted.
4. When the user has inserted enough money for the item, the user pushes a button.
5. The machine releases the item and deducts the price from the stored money.
6. The machine returns change. [Ours won’t.]
Components Needed for the General Protocol

What makes up the state of our vending machine?

Simplest answer: money stored.
Let’s use a register to record the amount of money.
When money is inserted, use an adder.
When a purchase is made, use a subtractor (that is, an adder).

What is the Unit of Money Stored?

How much do products cost? $1 to $2
How much money can the machine store?
Enough for a product, so $2 to $4.
Should we accept coins or bills or both?
Realistic answer: both.
Our answer: coins...but no pennies ($0.01)!
Let’s count money in nickels ($0.05).

How Big is the Register for Storing Money Inserted?

State is a register $N$, the number of nickels.

How many bits do we need for $N$?
The machine should store $2$ to $4$.
The value in $N$ is in units of $0.05$.
So $N$ should hold at most around 40 to 80.
Use a 6-bit register as an unsigned value.
The maximum is then 63, or $3.15$.

What about Item Prices?

Prices should be easy to change.
Instead of using fixed values, let’s use more 6-bit registers: $P_1$, $P_2$, and $P_3$.
Machine owner can set the prices.
Prices are also state, but we abstract them away.
Design the FSM assuming that
- prices are constant, but
- not known in advance (must read registers).
Abstract State Table Entries for Coin Insertion

Initial state is always \textit{STATE}^N

<table>
<thead>
<tr>
<th>input event</th>
<th>cond.</th>
<th>final state</th>
<th>accept coin</th>
<th>release product</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>always</td>
<td>\textit{STATE}^N</td>
<td>\text{x}</td>
<td>none</td>
</tr>
<tr>
<td>quarter inserted</td>
<td>\text{N &lt; 59}</td>
<td>\textit{STATE}^{N+5}</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>quarter inserted</td>
<td>\text{N ≥ 59}</td>
<td>\textit{STATE}^N</td>
<td>no</td>
<td>none</td>
</tr>
</tbody>
</table>

Abstract State Table for Product Selection

Initial state is always \textit{STATE}^N

<table>
<thead>
<tr>
<th>input event</th>
<th>cond.</th>
<th>final state</th>
<th>accept coin</th>
<th>release product</th>
</tr>
</thead>
<tbody>
<tr>
<td>item 1 selected</td>
<td>\text{N ≥ P}_1</td>
<td>\textit{STATE}^N \text{– P}_1</td>
<td>\text{x}</td>
<td>1</td>
</tr>
<tr>
<td>item 1 selected</td>
<td>\text{N &lt; P}_1</td>
<td>\textit{STATE}^N</td>
<td>\text{x}</td>
<td>none</td>
</tr>
</tbody>
</table>

Bits of Input and Output

Inputs include:
- coin inserted: a 3-bit value \(C = C_2C_1C_0\) (assume representation provided to us)
- product selection buttons:
  - one for each product: \(B_1, B_2, \text{and } B_3\)

Outputs include:
- coin accept \(A\) (1 means accept, 0 reject)
- item release signals: \(R_1, R_2, R_3\)

The Input Representation is Provided for Us

<table>
<thead>
<tr>
<th>coin type</th>
<th>value</th>
<th># of nickels</th>
<th>(C_2C_1C_0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(none)</td>
<td>N/A</td>
<td>N/A</td>
<td>110</td>
</tr>
<tr>
<td>nickel</td>
<td>$0.05</td>
<td>1</td>
<td>010</td>
</tr>
<tr>
<td>dime</td>
<td>$0.10</td>
<td>2</td>
<td>000</td>
</tr>
<tr>
<td>quarter</td>
<td>$0.25</td>
<td>5</td>
<td>011</td>
</tr>
<tr>
<td>half dollar</td>
<td>$0.50</td>
<td>10</td>
<td>001</td>
</tr>
<tr>
<td>dollar</td>
<td>$1.00</td>
<td>20</td>
<td>111</td>
</tr>
</tbody>
</table>
Outputs Correspond to Inputs in the Previous Cycle

In our class,
  * FSM outputs do not depend on input, so
  * the FSM cannot respond in the same cycle.
Instead, the FSM's outputs
  * are calculated based on state and inputs,
  * then stored for a cycle in flip-flops.
The coin mechanism designer must know that the accept signal comes in the next cycle.
These stored outputs are also state!

Our Abstract Model and I/O are Specified

We have an abstract model.
  We have I/O in bits.
  What's next?
  Complete the specification!

Let's Calculate the Size of Our FSM

How many bits of state do we have?
Ignoring prices, we have
  * a 6-bit register, and
  * four bits of stored output, so
  * a total of 10 bits, or 1024 states.
How many input bits do we have?
Three bits of coin, three buttons, so 6 bits.
1024 states, each with 64 arcs. Good luck!

Ignore Output “State” and Unused Input Combinations

Obviously, we need to simplify.
First,
  * four stored output bits do not affect our transitions, so we can ignore them.
  * Each STATE<N> thus represents 16 equivalent states.
Second, two bit patterns are unused in the C (coin) representation, so we need only 48 (8 × 6) arcs.
But 48 arcs × 64 states is still too much.
Choose a Strategy to Handle Multiple Inputs

How can we simplify further?
The abstract model has nine input events:
◦ no input,
◦ five types of coins, and
◦ three types of purchases.

Where do the other 39 arcs come from?
Multiple inputs!
Let’s choose a strategy to handle them.

Ignore Output “State” and Unused Input Combinations

Let’s prioritize input events strictly,
meaning that we ignore lower-priority events.
Our strategy is as follows:
◦ purchases have highest priority:
  item 3, then item 2, then item 1;
◦ coin type inputs are distinct,
  so they can’t occur at the same time.
Now we can write a complete next state table
(for a given set of prices).

Let’s Look at STATE50 with $P_3 = 60$, $P_2 = 10$, $P_1 = 35$

<table>
<thead>
<tr>
<th>$B_3$ $B_2$ $B_1$ $C_4C_1C_0$</th>
<th>final state</th>
<th>A</th>
<th>$R_3$</th>
<th>$R_2$</th>
<th>$R_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 x x xxx</td>
<td>STATE50</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 1 x xxx</td>
<td>STATE40</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0 0 1 xxx</td>
<td>STATE15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0 0 0 010</td>
<td>STATE51</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 0 0 000</td>
<td>STATE52</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 0 0 011</td>
<td>STATE55</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 0 0 001</td>
<td>STATE60</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 0 0 111</td>
<td>STATE50</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 0 0 110</td>
<td>STATE50</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Use a Priority Encoder to Resolve Conflicting Purchases

Purchases have priority, so start with those.
Item 3 has priority, then item 2.
We’ll use a priority encoder.
Given four input lines, a 4-input priority encoder produces
◦ a signal $P$ indicating that at least one input is active (1), and
◦ a 2-bit signal $S$ encoding the highest priority active input.
The Truth Table Requires Only a Few Lines

Let's write a truth table.

If $B_3 = 1$, no other inputs matter.

Similarly, if $B_3 = 0$, but $B_2 = 1$, the output is determined.

And so forth.

Solve K-Maps to Find Output Expressions

Next is $S_1$.

$S_1 = B_3 + B_2$

And, finally, $S_0$.

$S_0 = B_3 + B_2'B_1$
Implementation of a 4-Input Priority Encoder

Vending Machine FSM (Purchases Only)

Now, we're ready to design the FSM for purchases.

Register N, the number of nickels.

Priority encoder for purchases.

Vending Machine FSM (Purchases Only)

S output of priority encoder selects which price to deliver to adder.

Registers store negative prices, so \( \text{PRICE1} = -P \).

Adder subtracts price from \( N \), current money.

Vending Machine FSM (Purchases Only)

If \( R = 1 \), store difference as new \( N \). Otherwise, keep old \( N \) value.

\( R = 1 \) iff purchase was requested (\( P \)) AND machine has enough money (\( C_{\text{out}} \)).

\( C_{\text{out}} = 1 \) iff \( N \geq \text{price} \).
Release signals are stored in flip-flops and held high in next cycle.

If purchase approved (R), decode selected item (S) and allow release.

We Need to Know the Value of an Inserted Coin

We can't buy anything unless we insert coins!
There's already an adder that we can use:
- when a coin is inserted,
- add the current state $N$
- to the value of the inserted coin,
- and write the sum back to register $N$
- if the sum doesn't overflow.
But we don't have the value of an inserted coin.

Use Logic to Convert Coin Input Bits to Coin Value

Remember this table? Let's build a converter.

<table>
<thead>
<tr>
<th>coin type</th>
<th>value</th>
<th>$V_4V_3V_2V_1V_0$</th>
<th>$C_2C_1C_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(none)</td>
<td>N/A</td>
<td>00000</td>
<td>110</td>
</tr>
<tr>
<td>nickel</td>
<td>$0.05 $</td>
<td>00001</td>
<td>010</td>
</tr>
<tr>
<td>dime</td>
<td>$0.10 $</td>
<td>00010</td>
<td>000</td>
</tr>
<tr>
<td>quarter</td>
<td>$0.25 $</td>
<td>00101</td>
<td>011</td>
</tr>
<tr>
<td>half dollar</td>
<td>$0.50 $</td>
<td>01010</td>
<td>001</td>
</tr>
<tr>
<td>dollar</td>
<td>$1.00 $</td>
<td>10100</td>
<td>111</td>
</tr>
</tbody>
</table>

Solve K-Maps for Our Coin Value Module

Let's do the K-maps.

<table>
<thead>
<tr>
<th>$V_4$</th>
<th>$C_3C_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 x</td>
</tr>
<tr>
<td>1</td>
<td>0 0 1 x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$V_3$</th>
<th>$C_2C_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>0 0 0 x</td>
</tr>
<tr>
<td>C1</td>
<td>0 1 1 x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$V_2$</th>
<th>$C_2C_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>0 0 0 x</td>
</tr>
<tr>
<td>C1</td>
<td>0 1 1 x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$V_1$</th>
<th>$C_2C_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>0 0 0 x</td>
</tr>
<tr>
<td>C1</td>
<td>0 1 1 x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$V_0$</th>
<th>$C_2C_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>0 0 0 x</td>
</tr>
<tr>
<td>C1</td>
<td>0 1 1 x</td>
</tr>
</tbody>
</table>
Solve K-Maps for Our Coin Value Module

Let’s do the K-maps.

\[ V_1 = C_1' \]
\[ V_0 = C_2' C_1 \]

\((C_2 \oplus C_1\) is ok, too.\)

V0
\[
\begin{array}{c|c|c|c|c}
\hline
\text{C2} & \text{C1} & \text{C0} & \text{V0} \\
\hline
0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 \\
1 & 0 & 0 & x \\
1 & 0 & 0 & x \\
\hline
\end{array}
\]

V1
\[
\begin{array}{c|c|c|c|c}
\hline
\text{C2} & \text{C1} & \text{C0} & \text{V1} \\
\hline
0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 \\
1 & 0 & 0 & x \\
1 & 0 & 0 & x \\
\hline
\end{array}
\]

Implementation of the Coin Value Module

And we can implement as shown here.

Vending Machine Full Implementation

The blue elements extend the design from the earlier (purchase-only) version.

Zero-extended coin value calculation.

Mux selects price when purchase is requested \((P=1)\) or coin value \((P=0)\).

Calculation of coin accept signal \(A\): no purchase requested \((P=0)\) and adding coin’s value does not overflow \(N\) \((C_{out} = 0)\).

Accept signal is stored in flip-flop and held high for next cycle.
Vending Machine Full Implementation

The blue elements extend the design from the earlier (purchase-only) version.

State is now allowed to change in two cases: purchase allowed ($R = 1$) or coin insertion accepted ($A = 1$).