Foundations of processor design: finite state machines

EECS 370 – Introduction to Computer Organization – Winter 2015

Robert Dick, Andrew Lukefahr, and Satish Narayanasamy

EECS Department
University of Michigan in Ann Arbor, USA

A simple device

Custom controller for vending machine.

We could use a general purpose processor, but a custom controller will be faster, lower power, and cheaper to produce if we make enough of them. It will also be slower to design and more expensive in low volume.

Take money, vend drinks.

Input and Output

Inputs:

- coin trigger
- refund button
- 10 drink selectors
- 10 pressure sensors

Outputs:

- 10 drink release latches
- Coin refund latch

Operation of Machine

Accepts quarters only

All drinks are $0.75

Once we get the money, they can select a drink.

If they want a refund, release any coins inserted

No free drinks!

No stealing money!

Building the controller

- Finite State
  - Remember how many coins have been put in the machine and what inputs are acceptable
- Read-Only Memory (ROM)
  - Define the outputs and state transitions
- Custom combinatorial circuits
  - Reduce the size (and therefore cost) of the controller

Finite State Machines

- A Finite State Machine (FSM) consists of:
  - K states: S = {s1, s2, … ,sk}, s1 is initial state
  - N inputs: I = {i1, i2, … ,in}
  - M outputs: O = {o1, o2, … ,om}
  - Transition function T(S,I) mapping each current state and input to next state
  - Output Function P(S) [or P(S,I)] specifies output

Two Common State Machines

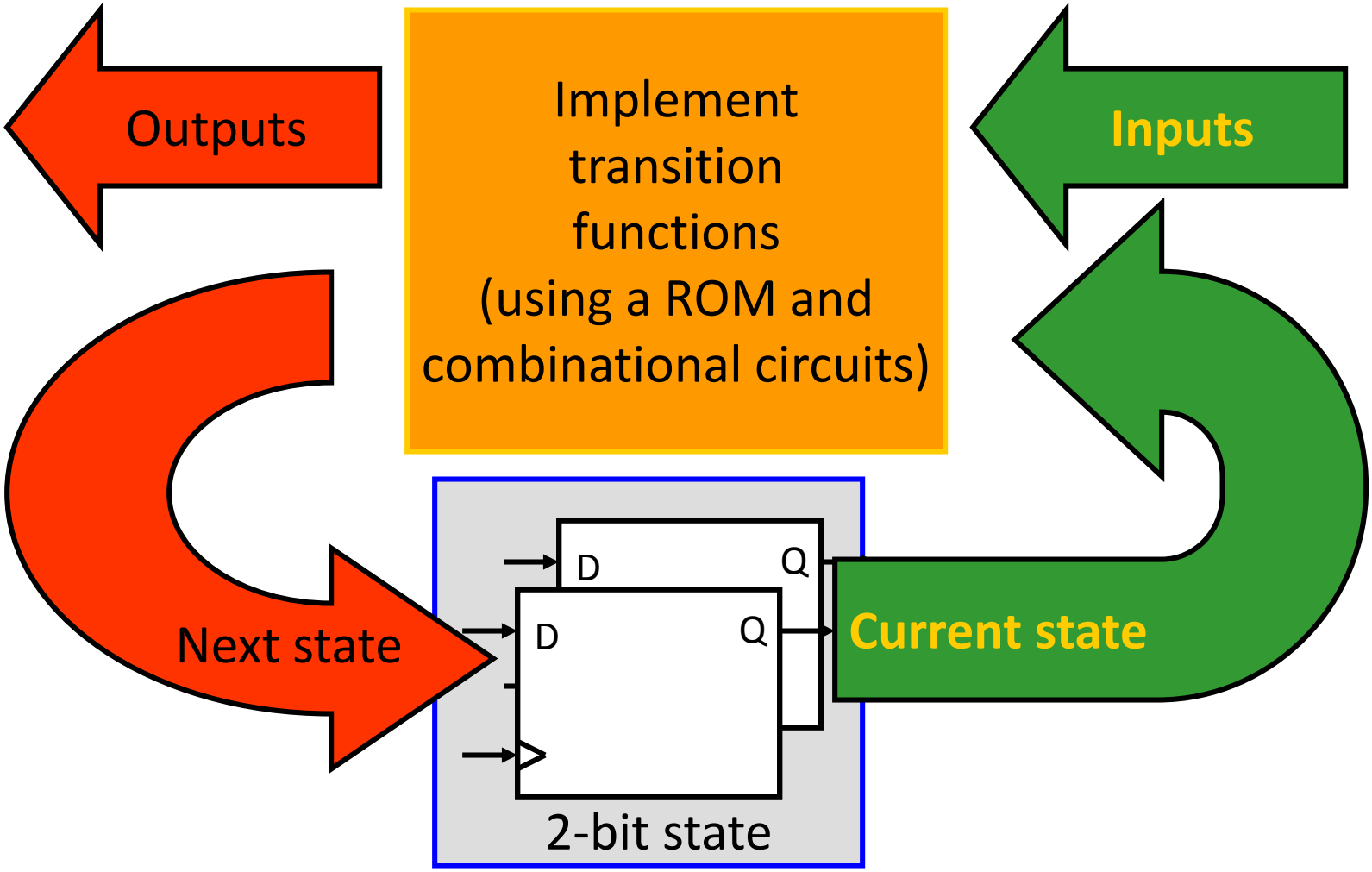
- Moore machine

  output function based on **current state** P(S) only

- Mealy machine

  output function based on **current state** and **current input** P(S,I)

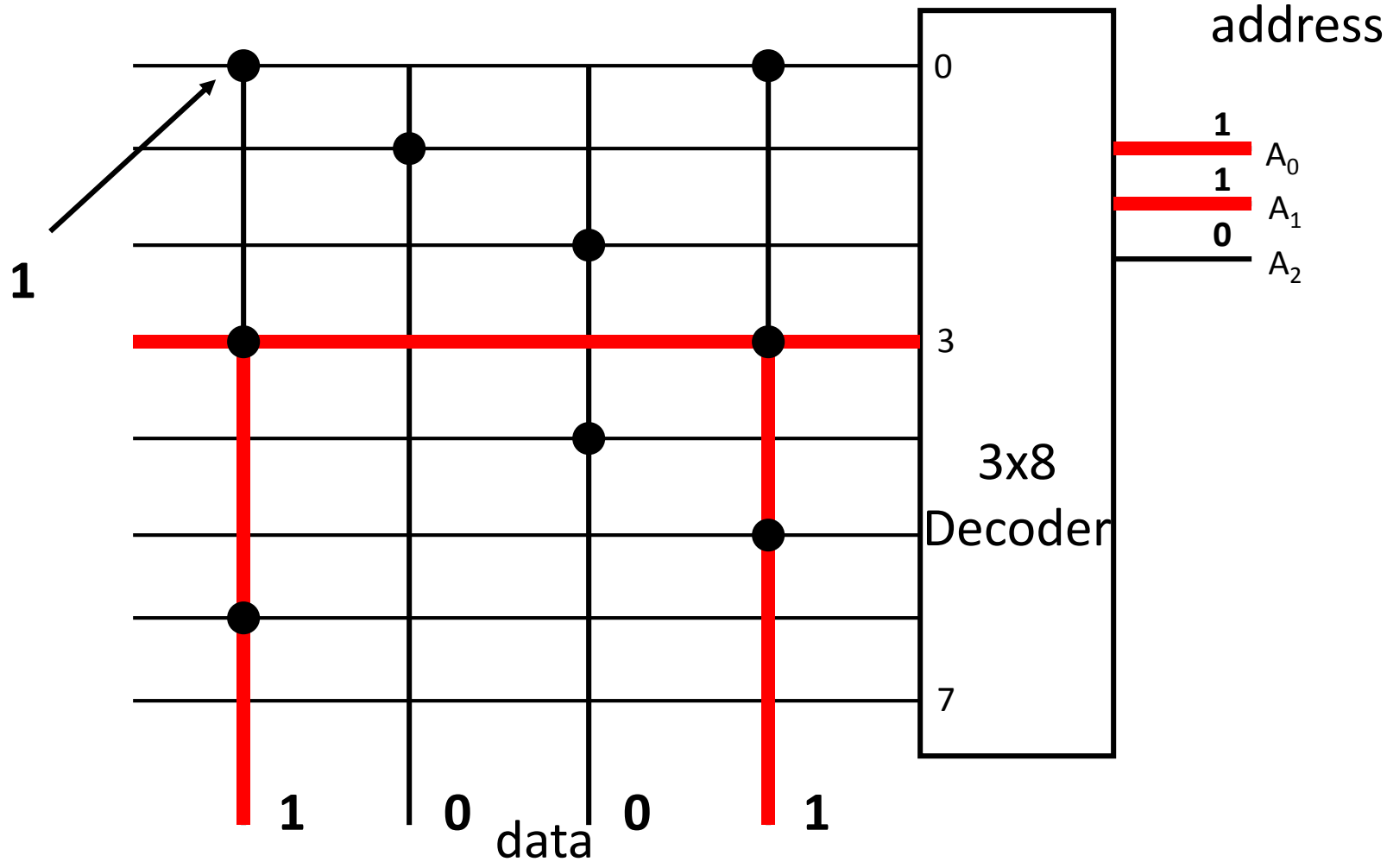
Implementing a FSM



ROMs and PROMs

- Read Only Memory
  - Array of memory values that are constant
  - Non-volatile
- Programmable Read Only Memory
  - Array of memory values that can be written exactly once (destructive writes)
- You can use ROMs to implement FSM transition functions
  - ROM inputs (i.e., ROM address): current state, primary inputs
  - ROM outputs (i.e., ROM data): next state, primary outputs

8-entry 4-bit ROM



ROM for Vending Machine Controller

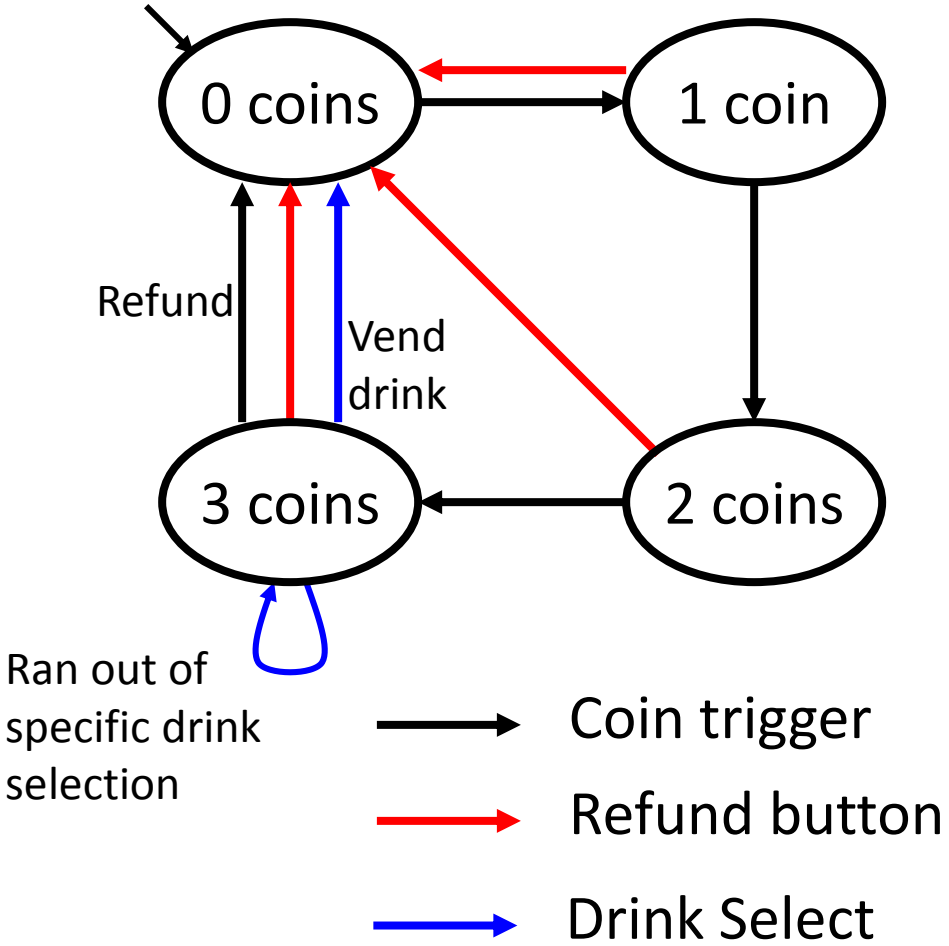
- Use current state and inputs as address
  - 2 state bits + 22 inputs = 24 bits (address)
  - Coin, refund, 10 drink selectors, 10 sensors
- Read next state and outputs from ROM
  - 2 state bits + 11 outputs = 13 bit (memory)
  - Refund release, 10 drink latches
- We need 2^{24} entry, 13 bit ROM memories
  - 218,103,808 bits! of ROM seems excessive for our cheap controller

Reducing the ROM needed

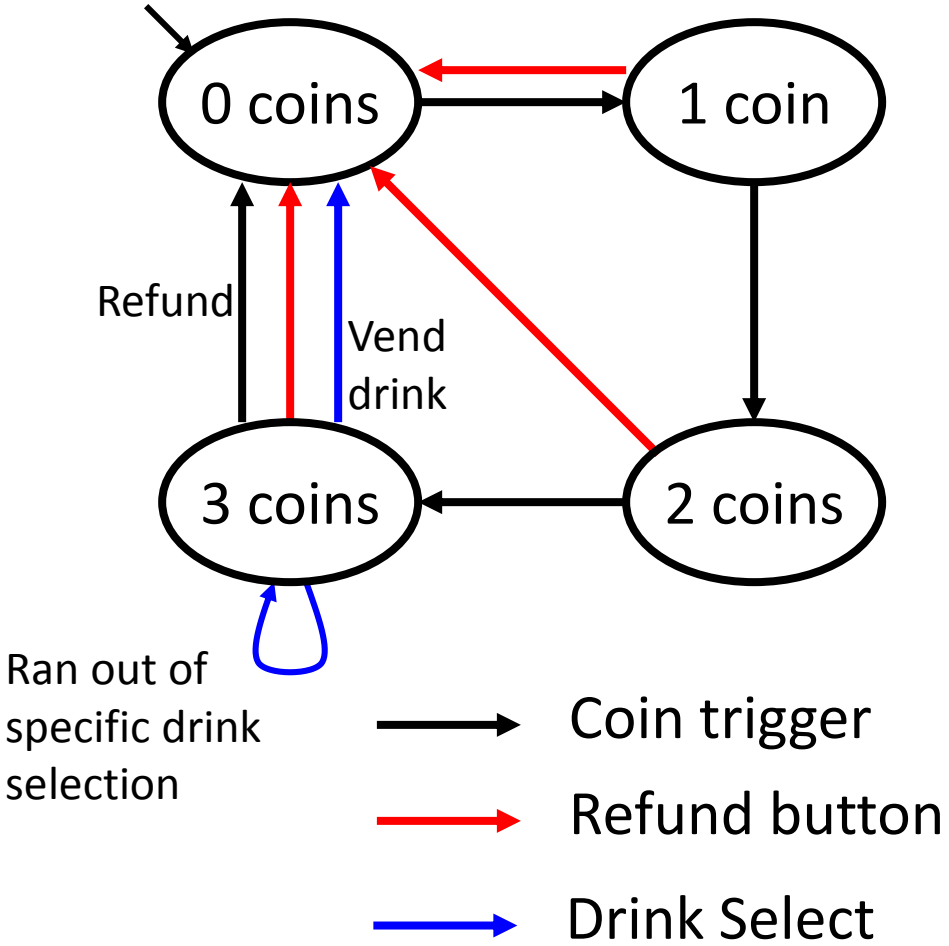
- Replace 10 selector inputs and 10 pressure inputs with a single bit input (drink selected)
  - Use drink selection input to specify which drink release latch to activate
  - Only allow trigger if pressure sensor indicates that there is a bottle in that selection. (10 2-bit ANDs)

- Now:
  - 2 current state bits + 3 input bits (5 bit ROM address)
  - 2 next state bits + 2 control trigger bits (4 bit memory)
  - $2^5 \times 4 = 128$ bit ROM (good!)

FSM for Vending Machine



FSM for Vending Machine



Some of the ROM contents

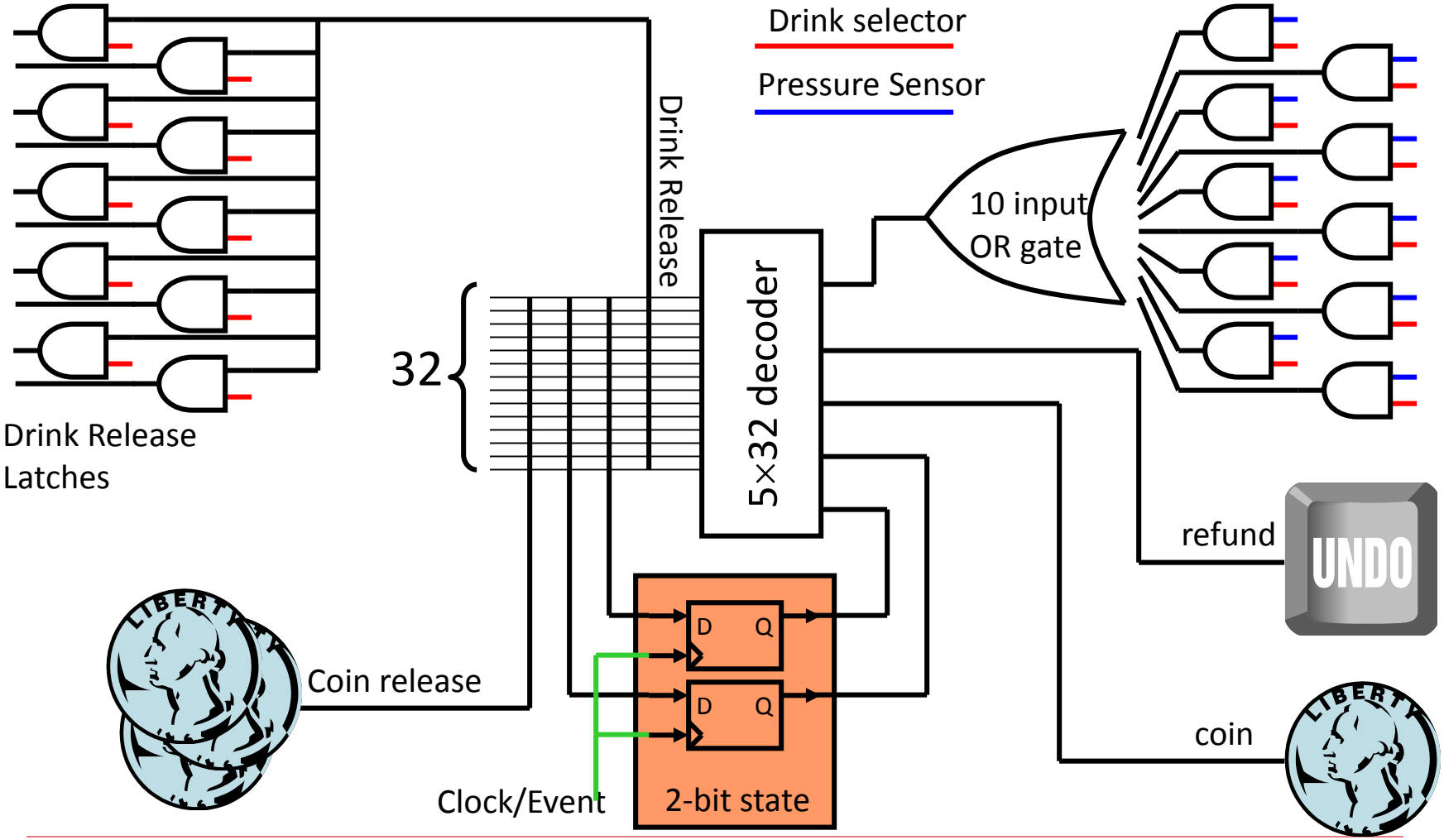
Current state	Coin trigger	Drink select	Refund button
0 0	0	0	0
0 0	0	0	1
0 0	0	1	0
0 0	1	0	0
0 1	1	0	0
1 0	1	0	0
1 1	0	1	0
1 1	1	0	0
… 24 more entries			

ROM address (current state, inputs)

Next state	Coin release	Drink release
… 24 more entries		

ROM contents (next state, outputs)

Putting it all together



Limitations of the controller

- What happens if we make the price $1.00?, or what if we want to accept nickels, dimes and quarters?
  - Must redesign the controller (more state, different transitions)
  - A programmable processor only needs a software upgrade.
    - If you had written really good software anticipating a variable price, perhaps no change is even needed

- **Next Topic - Our first processor!**