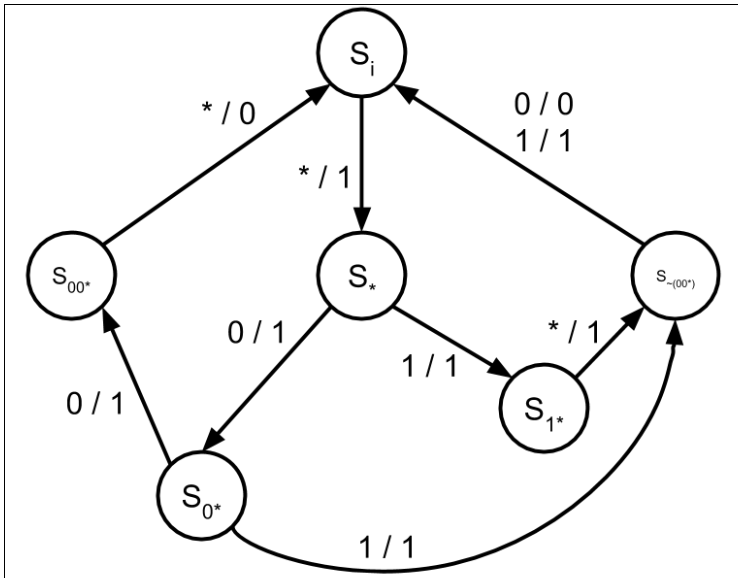
1. Optimal assignment of binary codes
    a. Was implementing binary code checker last time
        i. Previous lecture notes contain entire problem from start to finish
        ii. This section describes the binary code assignment portion we didn't get to in lecture
    b. Drew naïve implementation of FSM, then minimized it
        i. Minimized version below



State transition diagram

Present State	Next State		Output	
	x = 0	x = 1	x = 0	x =1
i	*	*	1	1
*****	0*	1*	1	1
0*	00*	~(00*)	1	1
1*	~(00*)	~(00*)	1	1
00*	*i*	*i*	0	0
~(00*)	*i*	*i*	0	1

State table

c. Will need $\lceil \log_2 6 \rceil = 3$ flip flops to represent 6 states
d. Can assign binary codes for states randomly
    i. Random assignment works
    ii. However, careful assignment reduces the combinational logic
e. Rule of thumb for state binary code assignments
    i. Try to assign adjacent (Hamming distance of 1) code words to a state and the state that follows it
    ii. If two states have the same next state, assign those states code words adjacent to next state
    iii. Creating a K-map helps immensely with this process
    iv. Initial state *i* will always be all 0s
f. Assign using the rules above
    i. Place i at 000, will always do this
    ii. Place * next to i at 010 since * is the successor to i
    iii. Place 0* and 1* adjacent to * at 110 and 011 respectively
    iv. 00* needs to be adjacent to i and 0*, which leaves 100 as the only place
    v. Would like ~(00*) to be adjacent to 0*, 1*, and i, but that isn't possible
        1. These are rules of thumb, not fixed laws
    vi. Can place ~(00*) at 001 to be adjacent to i and 1*, though
g. Note that there may potentially be more than one valid code assignment that minimizes distance

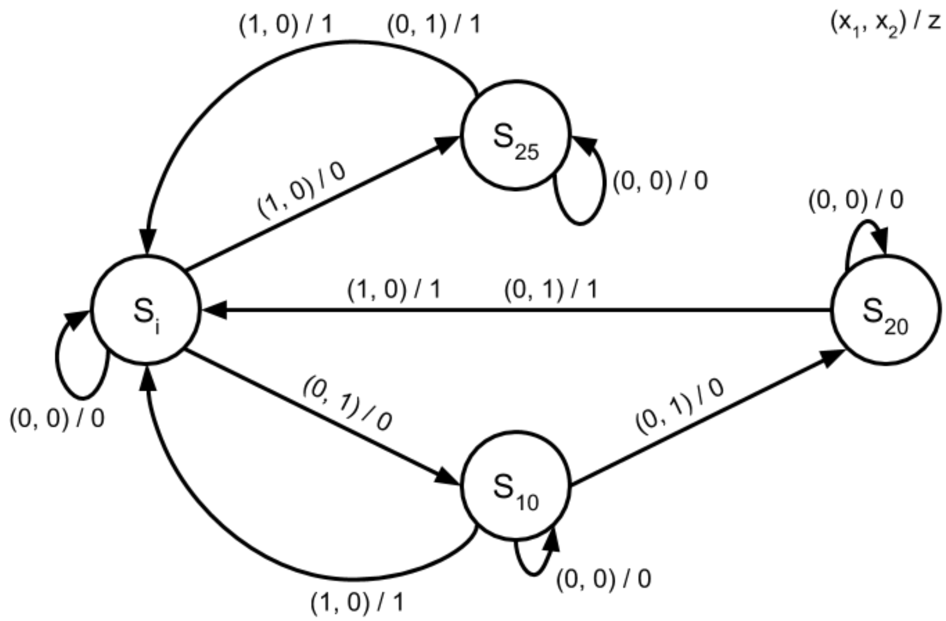
Binary Code AB

C		**00**	**01**	**11**	**10**
	0	*i*	*	0*	00*
	1	~(00*)	1*		

2. Debugging an FSM
    a. Generally, much more efficient to put in effort to get it right to begin with
        i. Students tend to jump straight to coding without planning program layout
        ii. Might work for smaller programs you write for classes… doesn't necessarily work in industry
        iii. Same idea here—put in time drawing FSM right to begin with, save time later
        iv. Nothing wrong with drawing a naïve FSM enumerating all states
            1. Once you've got that down, then minimize
            2. Trick is making sure that you draw that naïve FSM right to begin with
    b. One good way of seeing if your FSM you drew was right is to give a stream of inputs into your FSM
        i. Starting from your initial state *i*, give inputs and track your progression through your diagram
        ii. See what states you land at, and if those are what you expect
        iii. Also see what outputs you get, and if those are what you expect as well
    c. Examples
        i. With the BCD checker (both naïve and simplified) pass in 4 bit inputs and check output stream
            1. Outputs should be all 1s until you receive 4^{th} bit
            2. On 4^{th} bit, output 0 if input stream is 0-9 when interpreted as binary number
        ii. With the vending machine below, give a dime, then a dime, then a quarter
            1. Vending machine shouldn't vend before enough money inserted
            2. Change should be appropriate once enough money is inserted into machine

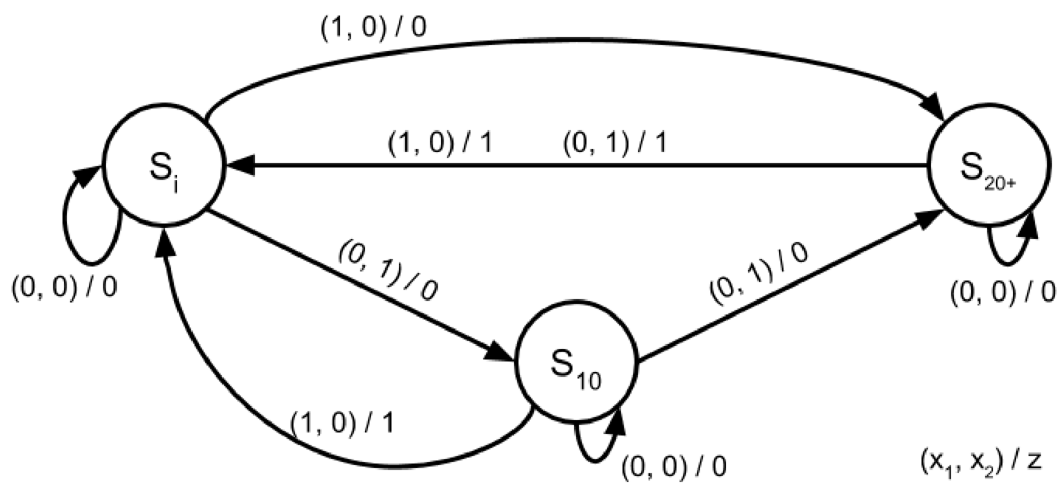
3. More complicated FSMs
    a. Design a vending machine that only takes dimes and quarters
        i. Merchandise is dispensed (z = 1) when the sum of the inputs ≥ 30 cents
        ii. Machine does not give change, no matter how much extra money gets deposited
    b. x_1 = quarter, x_2 = dime
        i. Two input, single output z
            a. State transition table will look somewhat different than you've seen before
        ii. Assume that it is not possible to input both quarters and dimes simultaneously
            a. How does that make sense physically?
        iii. No coin input results in no impact on machine
            a. What kind of vending machine eats your money if you don't deposit it fast enough?
    c. Will use a Mealy model
        i. Provides for simpler logic in the end for this case
        ii. As usual, though, a Moore model could provide simpler logic in certain cases

d. First, create state transition diagram
    i. Inputs are (x_1, x_2) for (quarter, dime)



e. Next, minimize the number of states using the Partition Minimization Procedure
    i. P_1 = (i, 10, 20, 25)
    ii. P_2 = (i) (10) (20, 25)
        1. 20 and 25 have same k-successors (i for both) so they stay together

f. Draw new state transition diagram, with new state called 20+



g. Assign code words next
    i. $\lceil \log_2 3 \rceil$= 2 flip flops
    ii. S_i starts in 00
    iii. No way to place all adjacent states 1 Hamming distance away
        2. Do the best we can, though

Binary Code

B \ A	**0**	**1**
0	*i*	20+
1	10	

h. Next, create state transition table
   i. Don't have to create state table since this is simple enough
   ii. Will do the same as BCD checker and add empty rows to the table for don't cares

Present State	Binary Code	Present State		Inputs		Next State		Output
		A	B	x_1	x_2	A'	B'	z
i	00	0	0	0	0	0	0	0
i	00	0	0	0	1	0	1	0
i	00	0	0	1	0	1	0	0
		0	0	1	1	d	d	d
10	01	0	1	0	0	0	1	0
10	01	0	1	0	1	1	0	0
10	01	0	1	1	0	0	0	1
		0	1	1	1	d	d	d
20+	10	1	0	0	0	1	0	0
20+	10	1	0	0	1	0	0	1
20+	10	1	0	1	0	0	0	1
		1	0	1	1	d	d	d
		1	1	0	0	d	d	d
		1	1	0	1	d	d	d
		1	1	1	0	d	d	d
		1	1	1	1	d	d	d

i. Finally, create K-maps from table above
   i. Be careful when entering values into K-map!
      1. For example, (A, B, x_1, x_2) = 0011 is missing since we can't have (x_1, x_2) = (1, 1)
      2. Make those inputs don't cares like normal
      3. We added extra rows to the binary code table
         a. This way, we know exactly where the don't cares go

A'

x_1x_2 \ AB	00	01	11	10
00	0	0	d	1
01	0	1	d	0
11	d	d	d	d
10	1	0	d	0

$$A' = A\overline{x_1}\,\overline{x_2} + Bx_2 + \bar{A}\bar{B}x_1$$

B'

x_1x_2 \ AB	00	01	11	10
00	0	1	d	0
01	1	0	d	0
11	d	d	d	d
10	0	0	d	0

$$B' = B\overline{x_1}\,\overline{x_2} + \bar{A}\bar{B}x_2$$

z

x_1x_2 \ AB	00	01	11	10
00	0	0	d	0
01	0	0	d	1
11	d	d	d	d
10	0	1	d	1

$$z = Ax_2 + Bx_1 + Ax_1$$