Sequential Logic Circuits; SDC Framework

CS 350: Computer Organization & Assembler Language Programming
Lab 8, due Fri Oct 25

[10/28: p.3 step 6 occurred twice]

A. Why?

- Storage elements are the basic circuits that store data, which is used in logic circuits that use memory.
- Finite state machines (FSMs) use sequential logic circuits and can model simple programs with fixed memory.
- Implementing the von Neumann architecture helps you understand how it works.

B. Outcomes

After this lab, you should be able to

- Design simple FSMs to recognize a simple pattern of characters or generate a simple pattern of output.
- Translate between FSMs state transition diagrams and tables and use them to trace FSM execution.
- Code (in C) the framework of a simulator for a simple von Neumann computer.

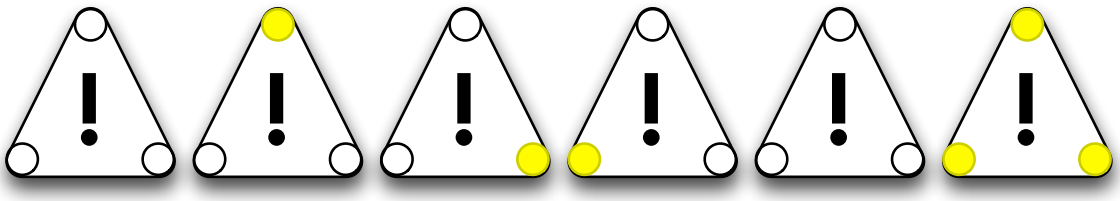
C. Written Problems [50 points total]

1. [15 points] Design a finite state machine that reads strings of `0`'s and 1's and accepts iff the input contains a `1011`. (The `1011` may appear anywhere inside the input; it can even be the whole input.) Give both state transition tables and diagrams and show a trace of execution on the input `01010110`, which should be accepted. (When giving the state transition table version of the machine, don't forget to also state what the start state is and what the accepting state(s) is/are.)
2. [15 points] Design a finite state machine that reads strings of `a`'s, `b`'s, and `c`'s and accepts iff each letter occurs an even number of times. For example,

`aaaa` and `abacbc` should be accepted but `aaa` shouldn't (the third `a` needs a mate); nor should `cabac` (the `b` is missing a mate). Note: 0 is an even number, so the empty string is an acceptable input.

As for the previous problem, give state transition table and diagram representations of your machine. Show traces of execution on the inputs `abacbc` and `cabac`. (Hint: you need at least 8 states.)

3. [20 points] Let's modify the flashing arrow example from the book so that we have a triangular sign with one light in each corner; clockwise from the top, let's call them lights 1, 2, and 3. When the switch is on, the lights flash in the order None, #1, #2, #3, None, All three, … (and repeat).



Give a finite state machine description for this sign (show both a state transition table and a state transition diagram), and include three bits of output with each state (one bit each for lights 1, 2, and 3). Also, give logic expressions that implement the combinatorial part of this machine. You can present them in full DNF form if you want (you don't have to optimize them). In any case, you don't have to draw them out as logic circuits.

D. Programming Problem [50 points total]

- For Labs 8 and 9, you'll be implementing a version of the Simple Decimal Computer (SDC) from lecture, in C. You'll write a line-oriented program that reads in the initial memory values and then lets the user execute the program one instruction at a time.
- The SDC is a decimal computer with memory addresses `0000` – `9999`, ten general-purpose registers numbered `0` – `9`, and ten instructions. The SDC uses word addressability, with a word being 4 decimal digits (plus a sign-magnitude sign).

- For Lab 8, we'll work on reading in memory and reading commands. You'll implement the actual instructions in Lab 9.

What Should Your Program Do?

1. Prompt for and read in the values for memory locations `00`, `01`, etc. Read until you see a number > `9999` or < `–9999`, and initialize the rest of memory to all zeros. (Each memory location is supposed to contain a value ≤ `9999` and ≥ `–9999`, so we can use values outside that range as sentinels.)
2. Print out the memory values and initialize the control unit. (For Lab 8, set the registers to zero and set the `done` flag to false.)
3. While not done
4. Prompt for and read a command (read the rest of the line including the carriage return).
5. For command `q`, set `done` to true.
6. For command `d`, dump out the control unit (program counter, instruction register, and data registers) and the memory values. (For Lab 8, just print out the registers (which are all zero).
7. For command `h` or `?`, print out a help message. (For Lab 8, this is just a placeholder message.)
8. For the null command (just a carriage return), call the `instruction_cycle` function to execute an instruction. (For Lab 8, this function just prints out a message saying how many times we've called it. It sets done to true the 10th time it's called.)
9. When you're done, dump the control unit and memory as for the d command.

E. Sample Solution and Skeleton

- To get you started with the program, there's a partial non-working skeleton `Lab08_skeleton.c`. Add, change, or delete lines in the skeleton as necessary; the STUB comments can be replaced with code. You don't have to use the skeleton if you don't want to, but you should understand how it works.

- There's also sample output in the file `Lab08_soln_out.txt`.
- I'll post a I'll post a sample executable solution on `alpha`. You can execute `~sasaki/Lab08_soln` .

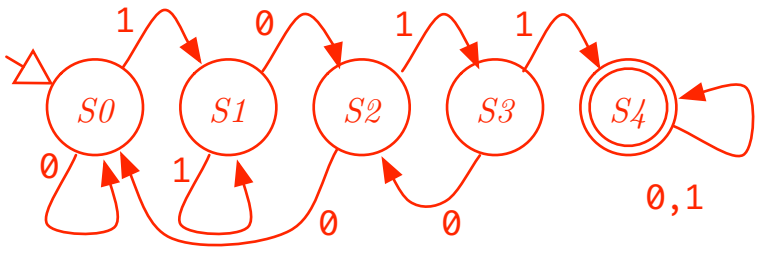
Solution to Written Problems

1. [Machine to accept `1011`]

State Transition Table (Start state = *S0*; Accepting state = *S4*)

State	*Input*	*New State*	*Comment*
S0	0	*S0*	*In S0 we've seen none of 1011*
S0	1	*S1*	
S1	0	*S2*	*In S1, we've seen 1*
S1	1	*S1*	
S2	0	*S0*	*In S2, we've seen 10*
S2	1	*S3*	
S3	0	*S2*	*In S3, we've seen 101*
S3	1	*S4*	
S4	0	*S4*	*Once we've seen 1011, we will accept regardless of the rest of the input*
S4	1	*S4*	

State Transition Diagram



Execution Trace

Execution ends in *S4*, which is an accepting state, so we accept the input.

State	*S0*	*S0*	*S1*	*S2*	*S3*	*S2*	*S3*	*S4*	*S4*
Input		0	1	0	1	0	1	1	0

2. [Machine to accept strings with an even number each of `a`'s, `b`'s, and `c`'s.] We need 8 states to model the $2 \times 2 \times 2$ different combinations of even and odd numbers of `a`'s, `b`'s, and `c`'s. I'll all the states `000`, `001`, …, `111` where the three bits indicate the parity of numbers of `a`'s, `b`'s, and `c`'s respectively. E.g., state `010` is for even `a`'s, odd `b`'s, and even `c`'s. State `000` is the starting and only accepting state.

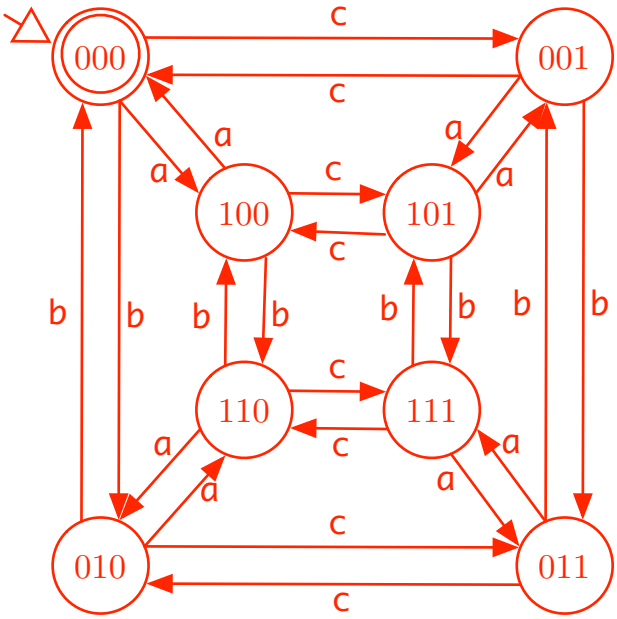
State Transition Table (Start state = 000; Accepting state = 000)

State	***Input***	***New State***
000	a	100
000	b	010
000	c	001
001	a	101
001	b	011
001	c	000
010	a	110
010	b	000
010	c	011
011	a	111
011	b	001
011	c	010

State	***Input***	***New State***
100	a	000
100	b	110
100	c	101
101	a	001
101	b	111
101	c	100
110	a	010
110	b	100
110	c	111
111	a	011
111	b	101
111	c	110

State Transition Diagram

Basically, the diagram can be arranged as a cube with two states in each dimension.



inputs `abacbc` and `cabac`. (Hint: you need at least 8 states.)

Execution Trace

Input `abacbc` is accepted:

State	000	100	110	010	011	001	000
Input		a	b	a	c	b	c

Input `cabac` is rejected because it has an odd number of `b`'s.

State	000	001	101	111	011	010
Input		c	a	b	a	c

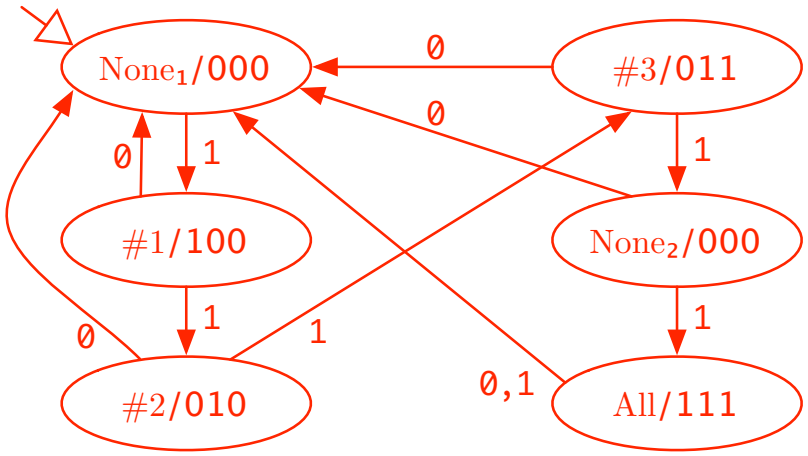
3. [Three-light traffic sign] There are six states, one for each part of the cycle of lights None, #1, #2, #3, None, All. The input is a sequence of `0`'s and `1`'s (for switch off and switch on). Each node below has a state/output combination (three bits of output specify which of lights 1, 2, and 3 are on). E.g., #3/`001` indicates that in state "#3", lights 1 and 2 are off and light 3 is on. There is no accepting state because we are only running the machine for its output.

State Transition Table

(Start state = None_1; no accepting states)

State	*Input*	*New State*
None_1	0	None_1
None_1	1	#1
#1	0	None_1
#1	1	#2
#2	0	None_1
#2	1	#3
#3	0	None_1
#3	1	None_2
None_2	0	None_1
None_2	1	All
All	0	None_1
All	1	None_1

State Transition Diagram



Note: We can also associate the light outputs with transitions. If so, then we want lights on if we're in the appropriate state and the input is on.

Logic circuit

For the logic gate circuit, we'll need to represent the states using bitstrings; let's use $000 = \text{None}_1$, $001 = \#1$, $010 = \#2$, $011 = \#3$, $100 = \text{None}_2$, and $101 = \text{All}$. (Since we have 6 states, we don't need 110 or 111.)

It's probably easiest to redraw the state transition table; we'll use bits S_2, S_1, S_0 for the current state and N_2, N_1, N_0 for the new state. For lights, let's use L_1, L_2, and L_3. The table below associates the light outputs with transitions.

Input X	*State*	S_2	S_1	S_0	*New State*	N_2	N_1	N_0	L_1	L_2	L_3
0	None_1	0	0	0	None_1	0	0	0	0	0	0
0	#1	0	0	1	None_1	0	0	0	0	0	0
0	#2	0	1	0	None_1	0	0	0	0	0	0
0	#3	0	1	1	None_1	0	0	0	0	0	0
0	None_2	1	0	0	None_1	0	0	0	0	0	0
0	All	1	0	1	None_1	0	0	0	0	0	0
1	None_1	0	0	0	#1	0	0	1	0	0	0
1	#1	0	0	1	#2	0	1	0	1	0	0
1	#2	0	1	0	#3	0	1	1	0	1	0
1	#3	0	1	1	None_2	1	0	0	0	0	1
1	None_2	1	0	0	All	1	0	1	0	0	0
1	All	1	0	1	None_1	1	1	0	1	1	1

Logic equations

- $N_2 = X\,\overline{S}_2\,S_1\,S_0 + X\,S_2\,\overline{S}_1\,\overline{S}_0 + X\,S_2\,\overline{S}_1\,S_0$
- $N_1 = X\,\overline{S}_2\,\overline{S}_1\,S_0 + X\,\overline{S}_2\,S_1\,\overline{S}_0 + X\,S_2\,\overline{S}_1\,S_0$
- $N_0 = X\,\overline{S}_2\,\overline{S}_1\,\overline{S}_0 + X\,\overline{S}_2 S_1\,\overline{S}_0 + X\,S_2\,\overline{S}_1\,\overline{S}_0$
- $L_1 = X\,\overline{S}_2\,\overline{S}_1\,S_0 + X\,S_2\,\overline{S}_1\,S_0$
- $L_2 = X\,\overline{S}_2\,S_1\,\overline{S}_0 + X\,S_2\,\overline{S}_1\,S_0$

- $L_3 = X\ \overline{S}_2\ S_1\ S_0 + X\ S_2\ \overline{S}_1\ S_0$
- If we associate light outputs with just states, not transitions, then we drop the input X from the logic equations above. (So we take $L_3 = X\ \overline{S}_2\ S_1\ S_0 + X\ S_2\ \overline{S}_1\ S_0$ and turn it into $L_3 = \overline{S}_2\ S_1\ S_0 + S_2\ \overline{S}_1\ S_0$.)

