# CSO-Recitation 13

## CSCI-UA 0201-007

R13: Assessment 11 & Assessment 12 & Lab5 & FSM

# Today’s Topics

- Assessment 11
- Assessment 12
- Lab5
- Sequential Logic, FSM

# Before we start…

- Lab5 due is postponed to Dec 14
- Lab6 is changed to be optional..
  - If completed, it gives you an extra 5% of your total grade
- A final-assessment on the final week
  - 5% of your total grade
  - just another assessment, not a final exam thing, don’t worry~

# Assessment 11

# Q1 Explicit list

Which of the following statements are true about explicit list?

A. The explicit list design explicitly chains together all chunks of the heap into a linked list.

B. The explicit list design explicitly only chains together all free chunks of the heap into a linked list.

C. The explicit list design incurs more memory overhead than the implicit list design because it uses extra space in the header to store the next/prev fields.

Only store next/prev pointers in free chunks

D. malloc(...) in the explicit list design is faster than that of implicit list because it does not need to scan over allocated chunks.

E. free(...) in the explicit list design is faster than that of implicit list because it does not need to scan over allocated chunks.

# Q2 Buddy system

A special case of segregated list:
- each freelist has identically-sized blocks
- block sizes are power of 2

allocate:
- Recursive split in half

free:
- Recursively merge

Which of the following statements are true about the buddy system?

A. All chunks have sizes that are powers-of-2.

B. During free(...), coalescing only happens once by merging the freed chunk with its buddy of the same size if the buddy is free.

C. During free(...), coalescing is done recursively by repeatedly merging the freed chunk with its buddy of the same size and repeating the merge process for the resulting larger free chunk until its buddy is no longer free.

D. The design maintains multiple free lists each of which contains free chunks of the same (powers-of-2) size.

E. The design maintains a single free list containing all free chunks.

# Q3 Lab4 implicit list

- This series of questions assume the implicit list design of Lab4. Chunks have headers but not footers. The header is of size 16-bytes, expressed in the following C type

```
typedef struct {
        size_t size;  //8-byte size of the chunk including header
        size_t allocated; // 8-byte allocation status. 0 means free, non-zero means allocated
} header_t;
```

# Q3.1 Heap consistency invariants

Which of the following must always be true (aka invariants) if the heap is correct and not corrupted?

A. The sizes of all chunks--as found by traversing the entire heap--must add up to mem_heapsize().

B. There are no consecutive free chunks in the heap, if the implementation merges a freed chunk with its next free chunk. possible, no merge with prev here

C. The address of the last byte of the last chunk--as found by traversing the entire heap--must be equal to mem_heap_hi().

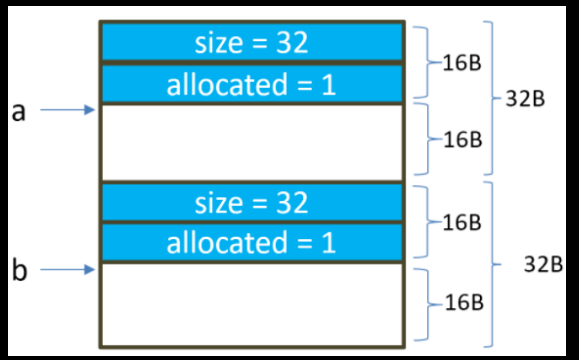
D. The size of every chunk must be multiples of 16.

E. The number of chunks found in the heap must be equal to the number of mm_malloc requests handled by the heap.

# Q3.2 Consequences of double free

- Assuming you have a fixed-size heap of 64 bytes. Furthermore, suppose the heap looks like the following after executing the first 2 lines of user code.

```
1: long *a = mm_malloc(sizeof(long)*2);
2: long *b = mm_malloc(sizeof(long)*2);
3: mm_free(b);
4: mm_free(a);
5: long *c = mm_malloc(sizeof(long)*6);
6: for (int i = 0; i < 6; i++) {
7:     c[i] = i;
8: }
9: mm_free(b);
```




- which of the following statement is true?

A. After executing line 4, the heap contains two free chunks of size 32 bytes each.

should do merge when free

B. After executing line 4, the heap contains one free chunk of size 64B.

C. After executing line 5, variable c has the same value as a.

D. After executing line 5, variable c is NULL.

# Q3.3 Consequences of double free, conti

```
void mm_free(void *p)
{
1:    header_t *h = payload2header(p);
2:    h->allocated = 0;
3:    mm_checkheap();
4:    coalesce(h);
}
```

Continuing from Q3.2, what happens during the execution of line 9 (user doubling free b)?

Assume that your implementation of `mm_free looks like follows:

A. mm_free returns immediately because it can notice that b has already been freed before.

B. Line 2 of mm_free would set c2 to zero.

C. Line 2 of mm_free would set c3 to zero.

D. Line 2 of mm_free would set c4 to zero.

E. mm_checkheap would fail because its heap traversal would find two chunks, one of whose size is not a multiple of 16.

F. mm_checkheap would fail because its heap traversal would find two chunks, the sum of whose sizes does not add up to mem_heapsize()

G. mm_checkheap would not fail because its heap traversal would find only one chunk, whose size is 64 and whose status is allocated.

# Q3.4 Consequences of overflow on the heap

This question assumes the following sequence of user code

```
1: long *a = mm_malloc(sizeof(long)*2);
2: long *b = mm_malloc(sizeof(long)*2);
3: for (int i = 0; i < 4; i++) {
4:     a[i] = i;
5: }
```

Assume the state of the heap after executing line 1 and 2 is the same as the picture in Q3.2. What of the following statements are true after finishing executing the loop (line 3-5)?

A. There will be a segmentation fault because the loop writes a2 and a3, both of which lies beyond the allocated space of 16 bytes.

B. The header of the chunk for the payload pointed to by b is overwritten by the loop.

C. The header of the chunk for the payload pointed to by b has size 2.

D. The header of the chunk for the payload pointed to by b has size 3. allocated status=3

E. If one invokes mm_checkheap() after finishing the loop, one would likely find that the heap is inconsistent (assuming uninitialized memory contains random values)

# Q4 Boolean expression equality

Which of the following Boolean expression equality holds?

A. $A+1=1$

B. $A+(B \cdot C)=(A \cdot B)+(A \cdot C)$ $A+(B \cdot C)=(A+B) \cdot (A+C)$

C. $A+A=A$

D. $A+A=1$

E. $A \cdot A=A$

F. $A \cdot A=0$

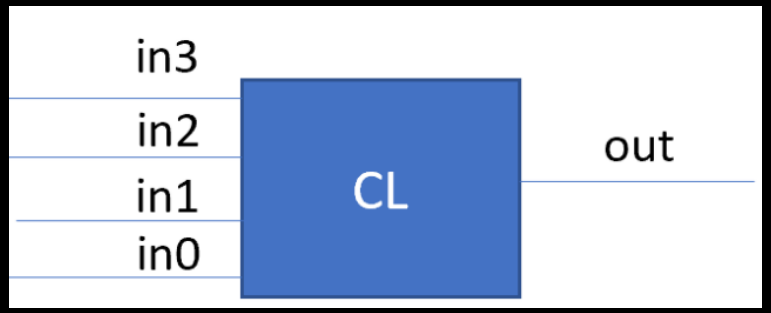
G. $\overline{A+B}=\bar{A} \cdot \bar{B}$

H. $\overline{A+B}=\bar{A}+\bar{B}$

I. $\overline{A \cdot B}=\bar{A}+\bar{B}$

J. $\overline{A \cdot B}=A \cdot B$

# Q5 Combinatorial circuit




- Suppose you are asked to build a combinatorial circuit to determine if an unsigned 4-bit integer is some multiple of 3. Note that 0 is considered to be a multiple of 3.
- In the above picture, the output signal out is 1 if and only if the 4-bit unsigned integer represented by in3, in2, in1, in0 (in0 is the least significant bit) is some multiples of 3.
- **Q5.1** Truth table
- What is the size of the truth table in terms of the number of rows needed to express the is-multiples-of-3 combinatorial circuit? To facilitate autograding, write your answer as a decimal number.
- 16

#row of truth table:

- have 4 input signals, each represents 1 bit
- how many bit patterns?
- 2^4 = 16

# Q5.2 From Truth Table to CL

- If you are to implement the is-multiples-of-3 combinatorial circuit as a sum (OR) of products (AND), how many AND gates do you need to use?
- 6
  - unsigned integer: 0~15
  - is multiple of 3:  0, 3, 6, 9, 12, 15
  - 6 AND gates

# Assessment 12

# Q1 Boolean laws

Which of the following Boolean laws hold? Below, A, B, C could refer to either a Boolean variable or a Boolean expression

A. R1: A+0=A
B. R2: A+0=0
C. R3: A+1=1
D. R4: A+1=A
E. R5: $A \cdot (B+C)=A \cdot B + A \cdot C$
F. R6: $A + \bar{A}=1$
G. R7: $A \cdot \bar{A} = 0$

Basic law:

- $A \cdot 0 =0$, $A \cdot 1=A$
- A+0=A, A+1=1

Distribution law

Inverse law

# Q2 Simplify boolean expression

- Simplify boolean expression (A+B) $\cdot$ ($\bar{A}$ + $\bar{B}$).
- You may write `*` for $\cdot$, and write `barA` for $\bar{A}$ (or `barB` or $\bar{B}$)
- (A+B)*(barA+barB)
- =(A+B)*barA + (A+B)*barB Distribution law
- =barA*a + barA*B + barB*A + barB*B Distribution law
- =0+barA*B+barB*A+0 Inverse law
- =barA*B+barB*A Basic law

# Q3 Simplify boolean expression

- When simplifying the Boolean expression in Q2, which of the Boolean laws in shown Q1 are needed?

A. R1
B. R2
C. R3
D. R4
E. R5
F. R6
G. R7

# Q4 Boolean circuit

- If you are to use a single logic gate to implement the simplified expression in Q2. Which gate should you use?

A. AND
B. OR
C. NOR
D. NAND
E. XOR
F. None of the above

NAND(A,B) = bar(A*B) = barA+barB

XOR(A,B) = A*barB + B*barA

# Q5 Gate delay

- What can be said about the gate delay (aka the number of gates along the longest path through a piece of logic) of an arbitrary combinatorial circuit built using sum of product representation? (Note: please ignore inverters when counting gate delays)

A. ≤2

B. ≥2

C. could be any delay

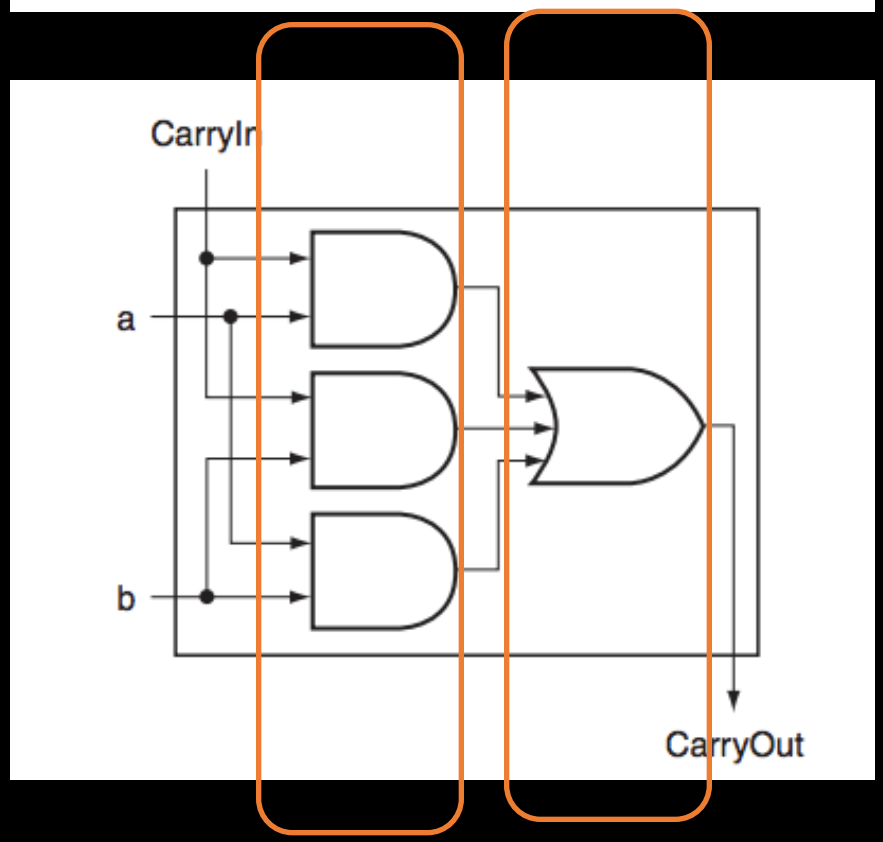
How to build a combinatorial circuit?

- truth table
- sum of product
- Gate delay?
  - "sum of product" – OR all ANDs
  - if no OR, no AND:
    - gate delay=0
  - if no OR:
    - gate delay=1
  - otherwise: gate delay=2

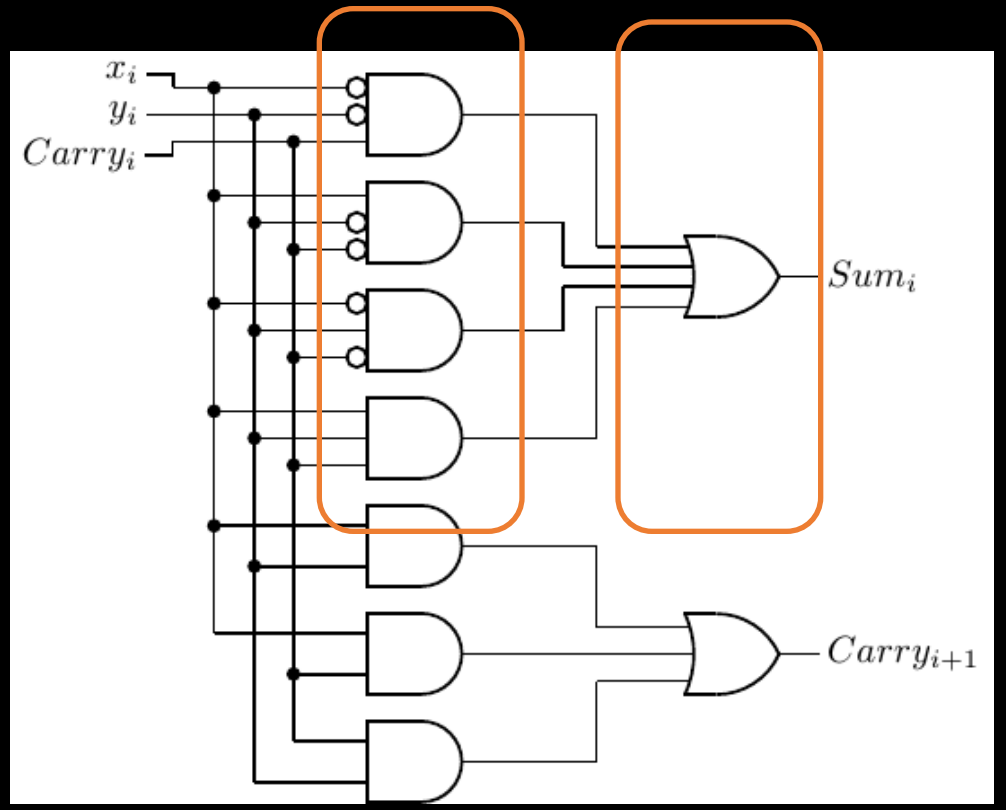
# Gate delay

- 1-bit adder:

$$\text{CarryOut} = (\text{b} \cdot \text{CarryIn}) + (\text{a} \cdot \text{CarryIn}) + (\text{a} \cdot \text{b})$$



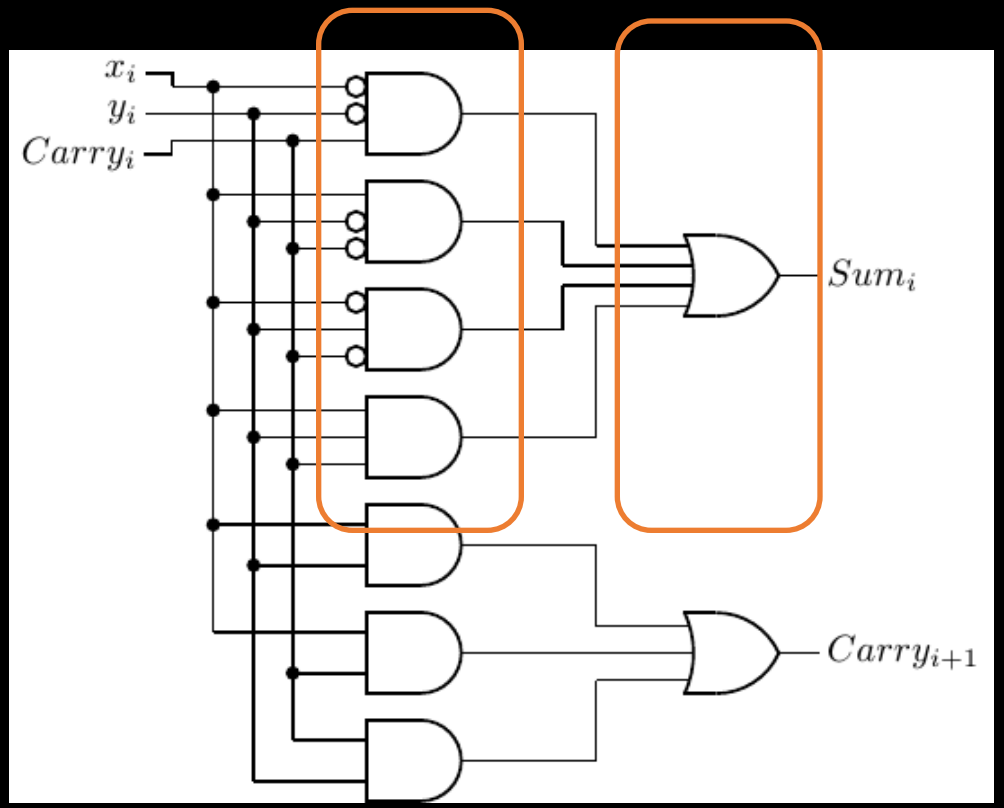

$$\text{Sum} = (\text{a} \cdot \overline{\text{b}} \cdot \overline{\text{CarryIn}}) + (\overline{\text{a}} \cdot \text{b} \cdot \overline{\text{CarryIn}}) + (\overline{\text{a}} \cdot \overline{\text{b}} \cdot \text{CarryIn}) + (\text{a} \cdot \text{b} \cdot \text{CarryIn})$$

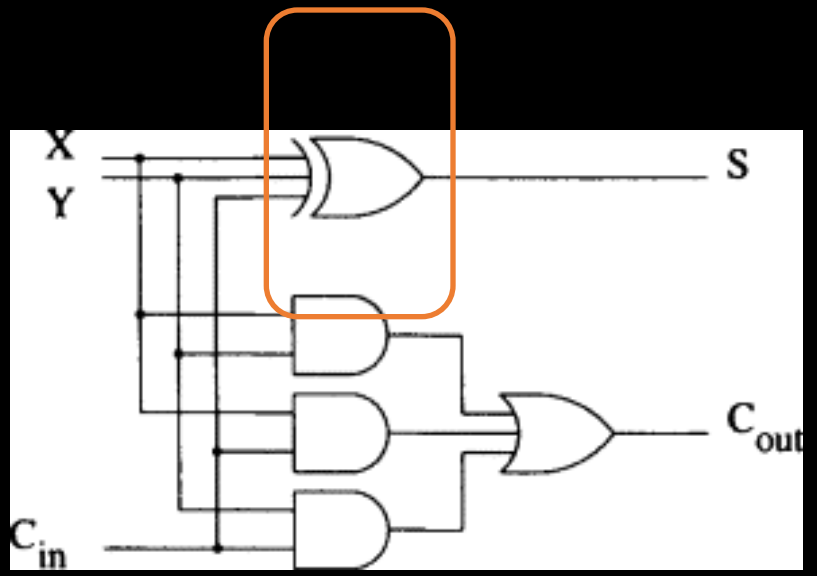


# Gate delay

- 1-bit adder:

$$\text{Sum} = (a \cdot \bar{b} \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot \bar{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$



# Ripple carry

- If a 1-bit adder's gate delay is 2, then what is the gate delay of a 32-bit ripple carry?
- To facilitate autograding, please write your answer as a single decimal number.
- 64

ripple carry:

- compute carry-bit one by one -> huge delay
- gate delay = gate_delay(1-bit adder) * #bits
- = 2* 32 =64

# Q7 Clocks

- Which of the following statements about the role of clocks in a digital system are correct?

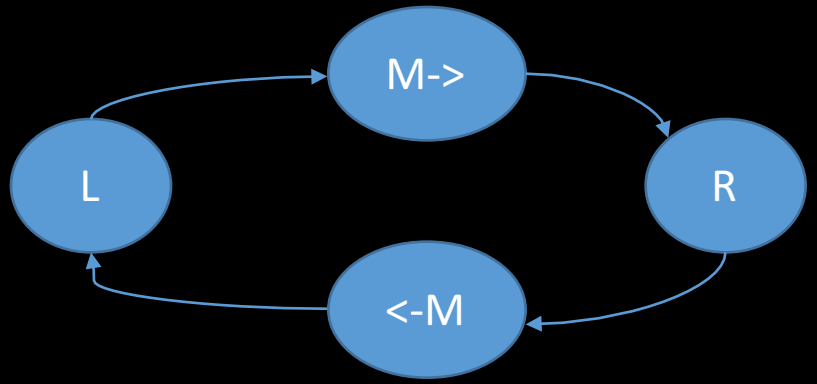
A. Clocks are needed for state elements.
B. Clocks are needed for combinatorial logic.
C. Clocks help determine when a state element is updated to a new value.
D. Clocks always run at 1GHz in all digital logic. clock frequency = 1/ clock_period
E. The longer the gate delay of the combinatorial circuit, the faster the clock can be used in the corresponding sequential logic.
F. The longer the gate delay of the combinatorial circuit, the slower the clock can be used in the corresponding sequential logic. delay of CL must be less than one clock cycle
G. The gate delay of the combinatorial circuit has no effect on the corresponding sequential logic.

# Q8 3 lights

- In the lecture example on "electronic eyes" (see slide 25 of https://nyu-cso.github.io/notes/arch-seq.pdf). The desired pattern of lights (left, middle, right light) to be lit up is: left, middle, right, middle, left, middle, right ...

- What is the minimum number of distinct state values needed for a FSM to implement this electronic eyes device?
- To facilitate autograding, please write your answer as a single decimal number.

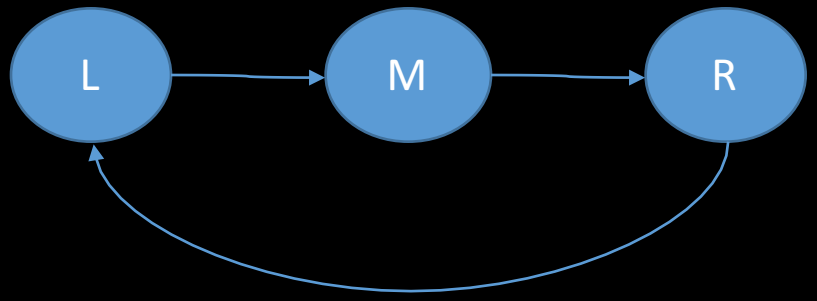
- 4



# Q9 3 lights, modified

- For the same lecture example on "electronic eyes", if the desired pattern of lights to be lit up is: left, middle, right, left, middle, right..., what is the minimum number of distinct state values needed for a FSM to implement this electronic eye device?

- 3



# Q10 3 lights, modified again

- For the same lecture example on "electronic eyes", if the desired pattern of lights to be lit up is: left, middle, middle, right, middle, middle, left, middle, middle, right, middle, middle..., what is the minimum number of distinct state values needed for a FSM to implement this electronic eye device?

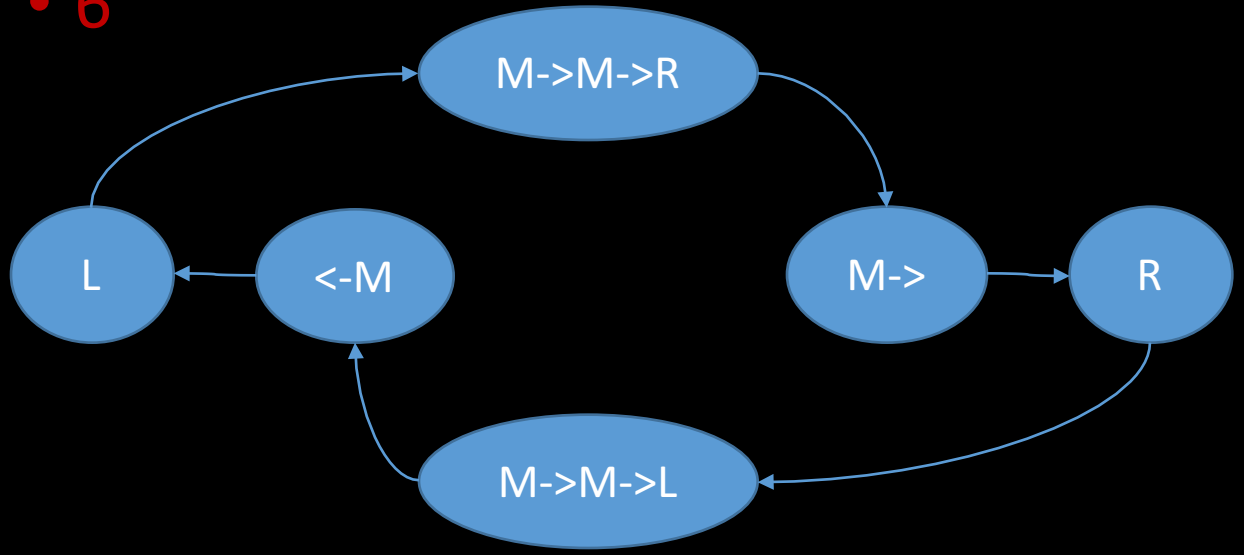
- 6

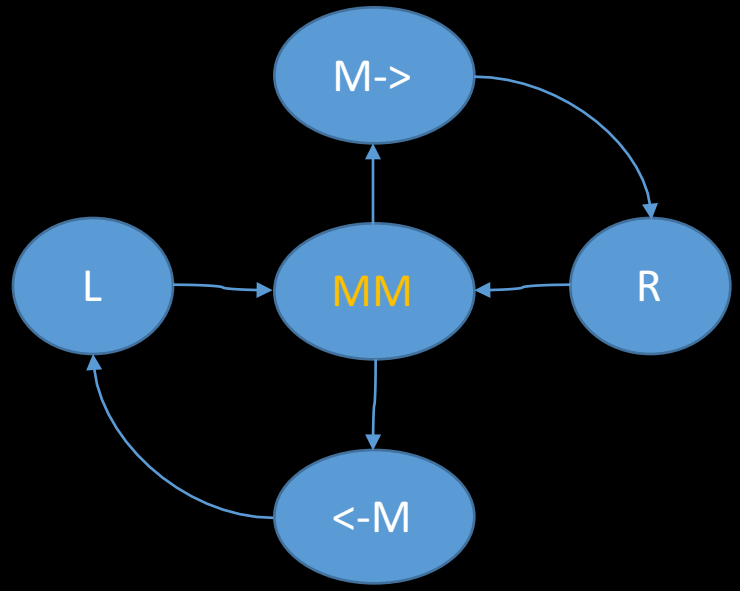
- If I'm standing in L now:
  - my next -> M
- If I'm standing in R:
  - my next -> M
- If I'm standing in M:
  - my next can be [R, L], or
  - My next can be M:
    - my next can be [R, L]

4

# Q10 3 lights, modified again

- For the same lecture example on "electronic eyes", if the desired pattern of lights to be lit up is: left, middle, middle, right, middle, middle, left, middle, middle, right, middle, middle..., what is the minimum number of distinct state values needed for a FSM to implement this electronic eye device?
- 6



# Lab5

using Logisim Evolution

# Setup

- Clone the Lab 5 repository – new link (campuswire)
- cd lab5new-<YourGithubUsername>
- wget https://nyu-cso.github.io/labs/logisim-evolution-2.15.jar
- download the Java runtime environment:
  - sudo apt-get install default-jre
- Launch Logisim by typing (in your lab5new repo):
  - java -jar logisim-evolution-2.15.jar

- Save frequently when you work~

# Part 2 exercise FSM

- There are 4 states
- what you write on the arrow edge is “transition condition”
- 0/1: bit before the backslash represents input, after the backlash is the output.
- it represents that at what condition we need to transit from which state to which state..

# Sequential logic

Building Blocks

# Sequential Logic

- There is memory
  - Outputs depend on prior state as well as the current inputs
  - State can be stored and used later
- We rely on clock signals
  - Clock signals tell us when things should happen
  - We should only write to state when the clock is set a certain way

# SR Latch

- Constructed from two NOR gates
  - You can either Set the latch (make it remember 1), or Reset it (make it remember 0)
- Two inputs: S and R
- Two outputs: Q and NOT Q
  - Q is what it remembers, NOT Q is the opposite
- Both S and R cannot be 1 at the same time, or sadness occurs

# D Latch

- Constructed from some additional logic and an SR Latch
- Two inputs: C and D
- You can have the latch remember D as long as C is true
- Two outputs: Q and NOT Q
  - Q is what the latch remembers, NOT Q is the inverse
- Ensures that S and R inputs to the SR Latch aren’t both true

# D Flip Flop

- Constructed from some additional logic and two D latches
- Same inputs and outputs as D latches
- But, the output is only stored on a chosen clock edge

# Finite State Machines

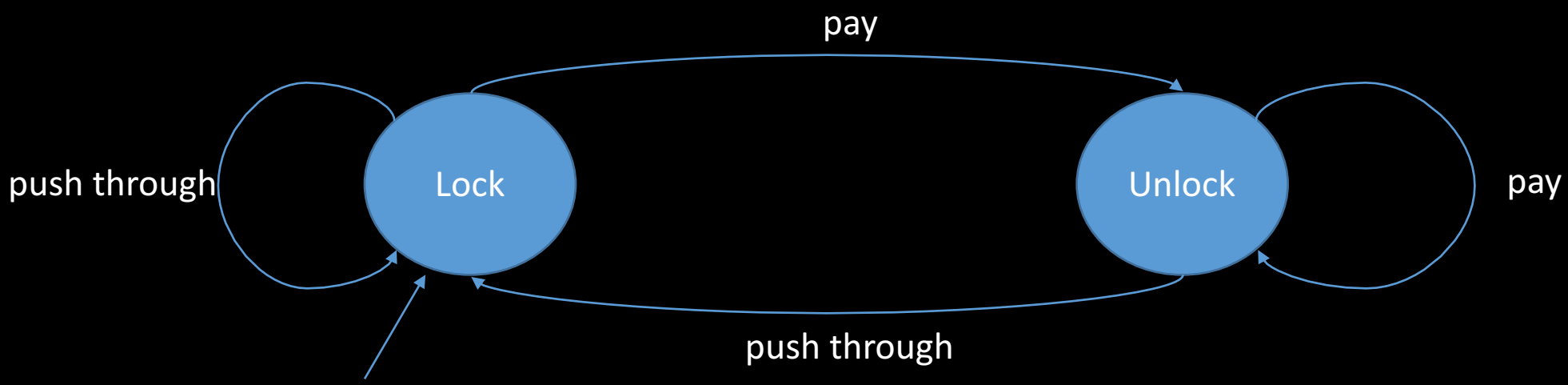
# Finite State Machines

- There are a number of states, inputs, and outputs
- To the beat of the clock, we read in inputs and go to new states, and set the outputs
- Both the output and the next state are defined by the current state and the inputs
- You can think of it as following a flow chart “when I’m on this step, and this is true, I go here”

# An FSM Example

- The NYC Subway Turnstile
- There is a lock controlled by the FSM
- If the user didn’t pay yet then the lock is active and the user can’t push through
- If the user pays the lock unlocks until they push through
- Draw an FSM for this
- Write out a truth table
- Create the circuit

# An FSM Example




| current state | input | next state |
| --- | --- | --- |
| (lock / unlock) | (pay / push through) | |