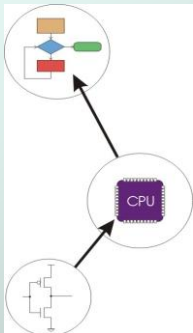


Chapter 3 Digital Logic Structures

Original slides from Gregory Byrd, North Carolina State University
Modified slides by Chris Wilcox, Colorado State University

Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.

Computing Layers



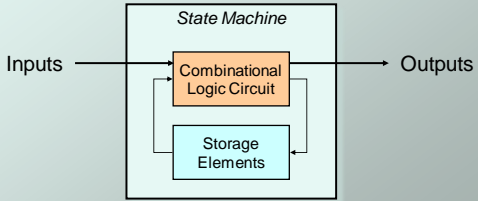
- Problems
-
- Algorithms
-
- Language
-
- Instruction Set Architecture
-
- Microarchitecture
-
- Circuits ←
-
- Devices

2

Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.

State Machine

- Another type of sequential circuit
 - Combines combinational logic with storage
 - “Remembers” state, and changes output (and state) based on **inputs** and **current state**

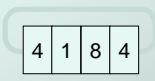


3

Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.


Combinational vs. Sequential

- Two types of “combination” locks



Combinational

Success depends only on the **values**, not the order in which they are set.



Sequential

Success depends on the **sequence** of values (e.g., R-13, L-22, R-3).

4

State

- The **state** of a system is a **snapshot** of **all the relevant elements** of the system at the moment the snapshot is taken.

Examples:

- The state of a basketball game can be represented by the scoreboard: number of points, time remaining, possession, etc.
- The state of a tic-tac-toe game can be represented by the placement of X's and O's on the board.

State of Sequential Lock

Our lock example has four different states, labelled A-D:

A: The lock is **not open**, and no relevant operations have been performed.

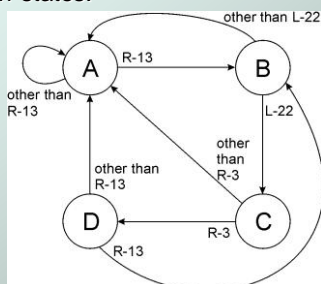
B: The lock is **not open**, and the user has completed the **R-13** operation.

C: The lock is **not open**, and the user has completed **R-13**, followed by **L-22**.

D: The lock is **open**.

State Diagram

- Shows **states** and **actions** that cause a **transition** between states.



Finite State Machine

- A system with the following components:

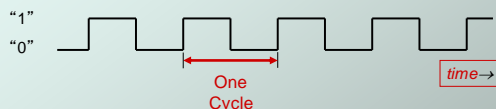
1. A finite number of **states**
2. A finite number of external **inputs**
3. A finite number of external **outputs**
4. An explicit specification of all **state transitions**
5. An explicit specification of what determines each external **output value**

- Often described by a state diagram.

- Inputs trigger state transitions.
- Outputs are associated with each state (or with each transition).

The Clock

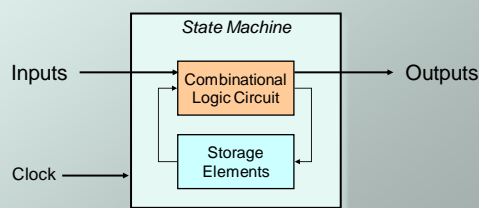
- Frequently, a **clock circuit** triggers transition from one state to the next.



- At the beginning of each clock cycle, state machine makes a transition, based on the current state and the external inputs.
 - Not always required. In lock example, the input itself triggers a transition.

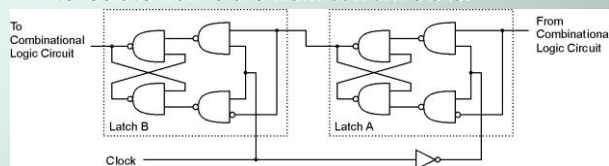
Implementing a Finite State Machine

- **Combinational logic**
 - Determine outputs and next state.
- **Storage elements**
 - Maintain state representation.



Storage: Master-Slave Flipflop

- A pair of gated D-latches, to isolate *next* state from *current* state.



During 1st phase (clock=1), previously-computed state becomes *current* state and is sent to the logic circuit.

During 2nd phase (clock=0), *next* state, computed by logic circuit, is stored in Latch A.

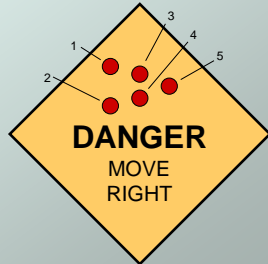
Storage

- Each master-slave flipflop stores one state bit.
- The number of storage elements (flipflops) needed is determined by the number of states (and the representation of each state).
- Examples:
 - Sequential lock
 - Four states – two bits
 - Basketball scoreboard
 - 7 bits for each score, 5 bits for minutes, 6 bits for seconds, 1 bit for possession arrow, 1 bit for half, ...

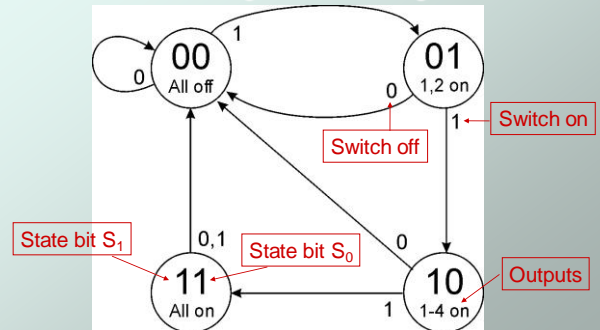
Complete Example

● A blinking traffic sign

- No lights on
- 1 & 2 on
- 1, 2, 3, & 4 on
- 1, 2, 3, 4, & 5 on
- (repeat as long as switch is turned on)



Traffic Sign State Diagram



Transition on each clock cycle.

Traffic Sign Truth Tables

Outputs
(depend only on state: $S_1 S_0$)

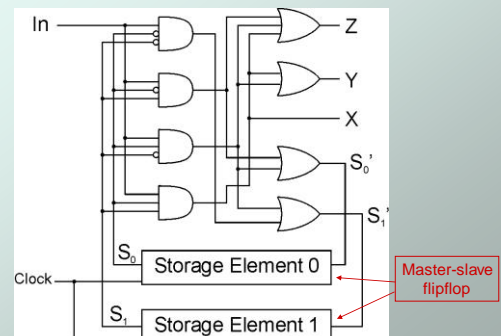
S_1	S_0	Z	Y	X
0	0	0	0	0
0	1	1	0	0
1	0	1	1	0
1	1	1	1	1

Next State: $S_1' S_0'$
(depend on state and input)

In	S_1	S_0	S_1'	S_0'
0	X	X	0	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	0	0

Whenever both are 1

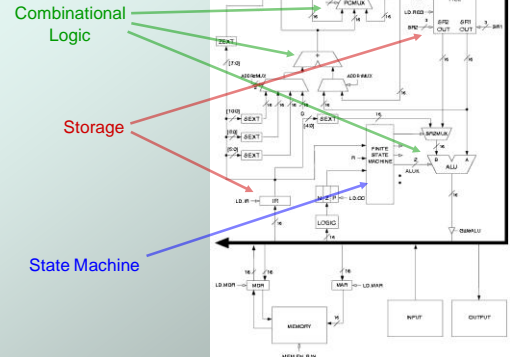
Traffic Sign Logic



From Logic to Data Path

- The data path of a computer is all the logic used to process information.
 - See the data path of the LC-3 on next slide.
- **Combinational Logic**
 - Decoders -- convert instructions into control signals
 - Multiplexers -- select inputs and outputs
 - ALU (Arithmetic and Logic Unit) -- operations on data
- **Sequential Logic**
 - State machine -- coordinate control signals and data movement
 - Registers and latches -- storage elements

LC-3 Data Path



Looking Ahead: C Arrays

- Array name can be used (and passed) as a pointer
- ```
// static allocation for array
int iArray[2] = {1234, 5678};
printf("iArray[0]: %d", iArray[0]);
printf("iArray[1]: %d", iArray[1]);
printf("&iArray[0]: %p", &iArray[0]);
printf("&iArray[1]: %p", &iArray[1]);
printf("iArray: %p", iArray);
```

## Looking Ahead: C Pointers

- Pointers can be used for array access
- ```
// dynamic allocation for array
int *iArray = malloc(2*sizeof(int));
iArray[0] = 1234; iArray[1] = 5678;
printf("iArray[0]: %d", iArray[0]);
printf("iArray[1]: %d", iArray[1]);
printf("&iArray[0]: %p", &iArray[0]);
printf("&iArray[1]: %p", &iArray[1]);
printf("iArray: %p", iArray);
```

Looking Ahead: C Structures

• Structures

```
struct Student {  
    char firstName[80];  
    char lastName[80];  
    int testScores[2];  
    char letterGrade;  
};  
struct Student student;  
struct Student students[10];
```

Looking Ahead: C Structures

• Structures

```
typedef struct _Student {  
    char firstName[80];  
    char lastName[80];  
    int testScores[2];  
    char letterGrade;  
} Student;  
Student student;  
Student students[10];
```

Looking Ahead: C Structures

• Structures

```
typedef struct {  
    char firstName[80];  
    char lastName[80];  
    int testScores[2];  
    char letterGrade;  
} Student;  
Student student;  
Student students[10];
```

Looking Ahead: C Structures

• Accessing structures

```
void func(Student student)  
{  
    strcpy(student.firstName, "John");  
    student.letterGrade = 'A';  
}  
  
void func(Student *student)  
{  
    strcpy(student->firstName, "John");  
    student->letterGrade = 'A';  
}
```


Looking Ahead: Makefiles

• File list and compiler flags

```
C_SRCS = main.c example.c
C_OBJS = main.o example.o
C_HEADERS = example.h
EXE = example

CC = c99
CC_FLAGS = -g -Wall -Wextra -c
LD_FLAGS = -g -Wall
```

Looking Ahead: Makefiles

• File dependencies

```
# Compile .c source to .o objects
.c.o:
    @echo "Compiling C source files"
    $(CC) $(CC_FLAGS) $<

# Make .c files depend on .h files
${C_OBJS}: ${C_HEADERS}
```

Looking Ahead: Makefiles

• Build target (default)

```
# Target is the executable
pa3: $(C_OBJS)
    @echo "Linking object modules"
    $(CC) $(LD_FLAGS) $(C_OBJS) -o $(EXE)
```

Looking Ahead: Makefiles

• Miscellaneous targets

```
# Clean up the directory
clean:
    @echo "Cleaning up project directory"
    rm -f *.o *~ $(EXE)

# Package up the directory
package:
    @echo "Cleaning up project directory"
    tar cvf r4.tar ../R4
```