ECE 124 - Digital Circuits and Systems

Andy Zhang

Winter 2014

Contents

1 Important Note **3**

2 Binary Logic **4**
- 2.1 Introduction . . . 4
- 2.2 Converting from One Base to Another . . . 4
- 2.3 Truth Tables . . . 4
- 2.4 Basic Logical Operations . . . 5
- 2.5 Properties of Boolean Algebra . . . 5
- 2.6 Circuit Costs . . . 5
- 2.7 Minterms and Maxterms . . . 5
- 2.8 Karnaugh Maps . . . 6
- 2.9 Karnaugh Map Elements . . . 6
- 2.10 NAND NOR . . . 6
- 2.11 XOR Gate . . . 6

3 Circuits **7**
- 3.1 Combinational Circuit . . . 7
- 3.2 Comparator Algorithms . . . 7
- 3.3 Decoders . . . 7
- 3.4 Encoders . . . 8
- 3.5 Multiplexers . . . 8
  - 3.5.1 Shannon Decomposition . . . 8
- 3.6 Sequential Circuits . . . 8
  - 3.6.1 Clock Signals . . . 8
- 3.7 Latches . . . 8
  - 3.7.1 SR Latch . . . 9
  - 3.7.2 SR Latch . . . 9
  - 3.7.3 SR Latch . . . 9
  - 3.7.4 SR Latch . . . 10
  - 3.7.5 Issues . . . 10
- 3.8 Flip Flops . . . 10
  - 3.8.1 DFF . . . 10
  - 3.8.2 Sets, Resets and Enables . . . 10
  - 3.8.3 Characteristic Tables of Flip Flops . . . 11
  - 3.8.4 Timing Analysis . . . 11
- 3.9 Registers . . . 11
  - 3.9.1 Parallel Load . . . 11

3.9.2 Universal Shift Registers . . . 11
3.10 Counters . . . 12
3.10.1 Ripple Counters . . . 12
3.10.2 Modulo Counters . . . 12

4 States **13**
4.1 Moore Machine . . . 13
4.2 Mealy Machine . . . 13
4.3 State Diagrams . . . 13
4.4 State Tables . . . 13

5 State Analysis and Design **14**
5.1 State Analysis Procedure . . . 14
5.2 State Design Procedure . . . 14
5.3 FF Impact on Design . . . 15
5.3.1 DFF . . . 15
5.3.2 TFF and JKFF . . . 15
5.4 Excitation Tables . . . 15
5.5 State Assignment . . . 15
5.5.1 Encoding . . . 15
5.6 State Reduction . . . 16
5.6.1 Implication Charts and Merger Diagrams . . . 16
5.6.2 Partitioning . . . 17
5.7 Algorithmic State Machine . . . 17
5.7.1 Design Steps . . . 17
5.7.2 One-Hot Encoding . . . 17
5.8 Asynchronous Sequential Circuits . . . 18
5.8.1 Definitions . . . 18
5.8.2 Analysis . . . 18
5.8.3 Design . . . 19
5.8.4 Reduction . . . 19
5.9 Races . . . 20
5.9.1 Race Condition . . . 20
5.9.2 Stability . . . 20
5.9.3 Race Free State Assignment . . . 20
5.10 Hazard and Output Glitches . . . 20
5.10.1 Definition . . . 20
5.10.2 Static Hazards . . . 21
5.10.3 Dynamic Hazard . . . 21
5.10.4 Fixing Hazards . . . 21
5.10.5 Hazards in Asynchronous Circuits . . . 21
5.10.6 Output Assignment in Asynchronous Circuits . . . 22

Chapter 1

Important Note

The content you will find in this review package contains mainly procedures on how to approach problems and theory. In order to fully understand the course content, it's highly recommended to do the practice problems. Many of the content here is taken from the slides of Andrew Kennings, some which are word for word, others which are summarized.

This document is better understood if the reader already has a general idea of what the course is about and it is not recommended to read this to learn the course content without prior knowledge.

Happy reading!

Chapter 2

Binary Logic

2.1 Introduction

Despite being able to count in base 10, computers aren't cool enough to do that. They only count in base 2. Consequently, to communicate with computers, we need to understand how base 2 works.

2.2 Converting from One Base to Another

The best way to explain this is through pseudocode:

```
$size \leftarrow \lceil \log_2 number \rceil$
$binary[size]$
$A \leftarrow -1$
$index \leftarrow size - 1$
**while** $A \neq 0$ **do**
    $binary[index] \leftarrow number\%2$
    $index \leftarrow index + 1$
    $A \leftarrow number/2$
**end while**
```

Example: $53 = (110101)_2$

2.3 Truth Tables

We define a logic function as a truth table. We exhaust all the different combinations of the input along with their output.

x	y	f
0	0	f_0
0	1	f_1
1	0	f_2
1	1	f_3

2.4 Basic Logical Operations

Here's a table of all the logic functions

Logic Operator	Symbol	Example
AND	$\cdot$, nothing	f_0
OR	$+$	f_1
NOT	$!, ', \neg, \bar{a}$	f_2

Please refer to `http://en.wikipedia.org/wiki/Logic_gate` for an image of each gate and their respective logic tables.

2.5 Properties of Boolean Algebra

- $x + 0 = x$, $x \cdot 1 = x$
- $x + x' = 1$, $x \cdot x' = 0$
- $x + x ==$, $x \cdot x = x$
- $x + 1 = 1$, $x \cdot 0 = 0$
- $(x')' = x$
- $x + y = y + x$, $x \cdot y = y \cdot x$
- $(x + y) + z = x + (y + z)$, $(x \cdot y) \cdot z = x \cdot (y \cdot z)$
- $x \cdot (y + z) = x \cdot y + x \cdot z$, $x + y \cdot z = (x + y) \cdot (x + z)$
- $(x + y)' = x' \cdot y'$, $(x \cdot y)' = x' + y'$
- $x + x \cdot y = x$, $x \cdot (x + y) = x$

2.6 Circuit Costs

NOT's are ignored, each gate costs 1 and each input costs 1

2.7 Minterms and Maxterms

Minterm:
For each row of the truth table, create an AND of the literals according to the following rule: If a variable has value 1 in the row, include its **+ve iteral**. If a variable x has a value 0 in the row, include its -ve literal. **Maxterm**:
For each row of the truth table, create an OR of the literals according to the following rule: If a variable x has the value 0 in the row, include its +ve literal. If a variable x has value 1 in the row, include its -ve literal.

We can then define a function as $f = m_a m_b m_c = m_d + m_e + m_f$

2.8 Karnaugh Maps

A Karnaugh Map contains all of the same information as a truth table but in a different form. The coordinates of each square determine the its value.
While labeling the rows and columns, it's important that only 1 value changes between adjacent rows and columns.

We can encompass groups of minterms via rectangles that are sizes of powers of 2(i.e. 1,2,4,8...). We can then represent that rectangle based on its coordinates that don't change. As a result, enclosing larger rectangles leaves less literals which is better.
Note that we can enclose minterms via the rectangles, from one side to another or adjacent.

Above 4 variables, we require more than 1 Karnaugh Map since we can only fit 4 variables in 1.

The same can be done with maxterms where you may enclose all of the 0s and create a Product of Sums.

Don't cares(X) are useful as they can be any value which can make minimizing a function simpler.

2.9 Karnaugh Map Elements

- A product term is called an **Implicant** if the logic function outputs a 1 for all minterms in the product term.
- An Implicant is called a **Prime Implicant** if the removal of any literal from the implicant results in a new product term that is not an implicant.
- A prime implicant is called an **Essential Prime Implicant** if it includes a minterm that is not found in any other implicant.

2.10 NAND NOR

OR Gate via NAND: $f = (x' \cdot y') = x + y$
NAND Gate via OR: $f = x' + y' + z' = x\bar{y}z$

When changing a circuit, a NOT on all inputs leads to a toggle between AND/OR and a NOT at the output. Ex: OR with all NOT input = NAND and vice versa.

2.11 XOR Gate

XOR gates are often hidden and can save a lot in terms of circuit costs.
$f_{XOR} = x_1 \oplus x_2 = \bar{x_1}x_2 + x_1\bar{x_2}$
$f_{XNOR} = x_1 \bar{\oplus} x_2 = \bar{x_1} \cdot \bar{x_2} + x_1 \cdot x_2$

Chapter 3

Circuits

3.1 Combinational Circuit

A combinatorial circuit is one that consists of logic gates with outputs that are determined entirely by the present value of the inputs.
Two operations we may perform:

- Analysis: given what is in the box, what function does it perform? We analyze by determining a function for the output starting from the input.
- Given functions to perform, what do we need in the box? We derive a truth table based on specification followed by simplifying it and implementing a circuit.

3.2 Comparator Algorithms

Equality: It's pretty straight forward. $A_i = B_i$ for any i. This can be done by letting $e_i = a_i b_i + a_i' b_i'$. If $e_i e_{i-1} ... e_2 e_1 = 1$, then $A = B$.
What if they weren't equal? We let $i = n$, the most significant bit(MSB). If $a_i > b_i$, then $A > B$ and we stop and vice versa. Otherwise, let $i = i - 1$. Stop once i=0 and all bits are considered.

$(A > B) = a_n b_n' + (e_n)(a_{n-1} b_{n-1}') + (e_n e_{n-1})(a_{n-2} b_{n-2}') + ... + (e_n e_{n-1} ... e_2 e_1 e_0)(a_0 b_0')$
$(A < B) = a_n' b_n + (e_n)(a_{n-1}' b_{n-1}) + (e_n e_{n-1})(a_{n-2}' b_{n-2}) + ... + (e_n e_{n-1} ... e_2 e_1 e_0)(a_0' b_0) = (A > B)'$

3.3 Decoders

When we have n bits, we can represent 2^n distinct patterns. We have **n-to-m decoders** with $m = 2^n$.
Often, some decoders have an enable signal. When enable = 1, then the decoder functions as desired, otherwise, all outputs = 0. For an image or more details, refer to `http://en.wikipedia.org/wiki/Decoder`
Smaller decoders become useful to implement larger ones.

3.4 Encoders

Encoders perform the inverse of a decoder. They have 2^n or fewer input and n output. If there are more than 1 high input, then there's an undefined output which leads to priority encoders which gives priority to higher numbered inputs.

3.5 Multiplexers

Multiplexers have data inputs, select inputs and single data outputs. The output data is based on the select input and the input data. Refer to `http://en.wikipedia.org/wiki/Multiplexer` for images and more detailed explanation.
Multiplexers are useful in many ways and can also be used to implement a n input function using a $n-1$ input multiplexer. Based on each pair of output, we can determine the input for the multiplexer.

3.5.1 Shannon Decomposition

Breaking a function down for a MUX implementation is called **Shannon Decomposition**
This works because given a boolean function $f = f(x_0, x_1...x_n)$, we can split the function into to as the following:

$$f = f(x_0, x_1...x_n) = x_0' f(0, x_1, x_2...x_n) + x_0 f(1, x_1, x_2...x_n)$$

3.6 Sequential Circuits

Unlike combinatorial circuits, the output of a sequential circuit also depends on the current state of the circuit. We define two types of sequential circuits:

- Asynchronous Circuits: circuit behaviour is determined by signals at **any instant in time and the order in which input signals change**
- Synchronous Circuits: circuit behaviour is determined from the knowledge of signal values at **discrete instances of time**

3.6.1 Clock Signals

A **clock signal** is used to control the behaviour of a circuit at discrete instances in time. They are periodic signals.
When the signal transitions from $0 \to 1$, we call it a rising edge. If it's $1 \to 0$, we call it a falling edge.

3.7 Latches

Latches are a storage element. They perform when a control signal is at 0 or 1 and not at any changing edge. This element is extremely useful to understand flip flops.
There exists several types of latches(`http://en.wikipedia.org/wiki/Flip-flop_(electronics)` for more detail and images)

3.7.1 SR Latch

S	R	Q	$\bar{Q}$
1	0	0	1
1	1	0	1
0	1	1	0
1	1	1	0
0	0	1	1

- When $S = 0$, $R = 1$, the output is $Q = 1$, and the circuit is in the **set state**
- When $S = 1$, $R = 0$, the output is $Q = 0$, and the circuit is in the **reset state**
- When $S = 1$, $R = 1$, it implies the reset state
- When $S = 0$, $R = 0$, the output holds at its previous value(storage)

3.7.2 SR Latch

S	R	Q	$\bar{Q}$
1	0	0	1
1	1	0	1
0	1	1	0
1	1	1	0
0	0	1	1

- When $S = 1$, $R = 0$, the output is $Q = 1$, and the circuit is in the **set state**
- When $S = 0$, $R = 1$, the output is $Q = 0$, and the circuit is in the **reset state**
- When $S = 0$, $R = 0$, it implies the reset state
- When $S = 1$, $R = 1$, the output holds at its previous value(storage)

3.7.3 SR Latch

C	S	R	next value of Q
0	X	X	hold
1	0	0	hold
1	0	1	0(reset)
1	1	0	1(set)
1	1	1	undefined

- When the control input $C = 0$, both latches are in hold state
- When the control input $C = 1$, the latch acts like an SR Latch(active high)

3.7.4 SR Latch

C	D	next value of Q
0	X	hold
1	0	0
1	1	1

In previous latches, we had the undesirable situation where the output was undefined due to both outputs being the same. To avoid this, we construct a latch where S and R can never be the same where $S = R'$

3.7.5 Issues

Latches don't allow precise control since they're only level sensitive. This creates an interval in time where the output can change rather than an instant. It would be better if the output can only change at an instant for more precise control.

3.8 Flip Flops

Unlike latches, flip flops allow more precise control on the output by letting the output change only during triggering i.e. changing edges.

3.8.1 DFF

Refer to `http://en.wikipedia.org/wiki/Flip-flop_(electronics)#D_flip-flop` for a much better explanation of how DFF works. Here's a rough explanation:
A DFF consists of 2 D Latches connected in series where the output of the first is the D input of the second. The clock input of each are inverses of the other.
While $CLK = 1$, the output of the first latch will follow the D input. When $CLK = 0$, the output will be disconnected from the first latch and connected to the second since $CLK = 1$ for the second(since the clocks are inverses of each other). The output of the second is then changed when the clock is triggered.

Depending on the type of trigger the flip flop depends on, we can position the inverter of the clocks on different inputs i.e. either the first or the second latch.

3.8.2 Sets, Resets and Enables

Flip flops can have additional control signals that will force Q to a known value.

- An asynchronous signal that forces $Q = 1$ is called an asynchronous **set** and remain that way if Q stays 1
- An asynchronous signal that forces $Q = 0$ is called an asynchronous **reset** and remain that way if Q stays 0
- A signal that prevents the clock from causing changes in Q according to D is called a clock enable

3.8.3 Characteristic Tables of Flip Flops

DFF

D	$Q(t+1)$
0	0
1	1

$Q(t+1) = D$

TFF

T	$Q(t+1)$
0	$Q(t)$
1	$Q'(t)$

$Q(t+1) = TQ'(t) + T'Q(t) = T \oplus Q(t)$

JKFF

J	K	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$Q'(t)$

$Q(t+1) = JQ'(t) + K'Q(t)$

3.8.4 Timing Analysis

In reality, it takes time for gates to change their outputs. There exist 3 timing parameters that are especially important:

- Setup Time(TSU): time it takes for data inputs to be stable prior to arrival of an active clock edge
- Hold Time(TH): time it takes for the data inputs need to be held stable after the arrival of the active clock edge
- Clock-To-Output Time(TCO): time it takes for the output to become stable after the arrival of the active clock edge

3.9 Registers

A group of n flip flops store n bits and is called an n-bit register. Each flip flop has a clear input and are connected to the same clear input. If clear = 0, all outputs are forced to 0.

3.9.1 Parallel Load

If load=1, the input data gets loaded into the input of the latch. Otherwise, the data output is fed back into the input.

3.9.2 Universal Shift Registers

Shift registers are able to shift data in a direction. As the active clock edge arrives, the data gets shifted to the next flip flop, resulting in a shifted data.

If we wanted a Universal Shift Register, we use multiplexers to perform multiple operations such as clear, load, shift left/right.
Universal Shift Registers have:

- an asynchronous clear signal
- a clock signal

- data inputs for parallel load
- data inputs for both left and right shifts
- two control inputs that determine the behaviour of the multiplexers

3.10 Counters

A counter is a register whose outputs go through a prescribed sequence of states upon the arrival of the **active edge of some triggering signal**(like a clock)
They can be either **asynchronous** or **synchronous**

3.10.1 Ripple Counters

Ripple counters consist of a series of of flip flops where the output Q of one flip flop is used as the clock of the next flip flop. Due to the lack of a common clock signal, this counter is asynchronous.

A binary ripple counter for example can be implemented easily using TFFs. For an up counter, Each bit depends on the falling edge of the previous bit where the first bit is always toggling.

The problem with feeding the output as a clock is that the counter is asynchronous which leads to timing issues. This is why we'd prefer synchronous counters by feeding the output to an AND gate with the previous input and all TFFs have a common clock signal.

3.10.2 Modulo Counters

If we don't want to count through all of the numbers but rather between an interval of numbers. We can use additional circuitry to detect the maximum count number and use a parallel load to restart the counting.

Chapter 4

States

4.1 Moore Machine

If the circuit output only depends on the current state, we call it a Moore Machine. It's important to note that in a Moore Machine, **outputs change synchronously** with the clock. The state also changes synchronously with the clock. This means the output changes at **predictable times**

4.2 Mealy Machine

If the circuit output depends on the current state and the current input values, we call it a Mealy Machine. In a Mealy Machine, outputs change asynchronously with respect to the clock since inputs can change at any time. This means the output can change at **unpredictable times**

4.3 State Diagrams

Sequential circuits are often depicted via a state diagram. These diagrams include:

- Possible states of the circuit
- Possible transition between states which depends on inputs
- Circuit output values

In a Moore Machine, recall that the circuit outputs are a function of the current state only. Consequently, the circuit output can be labeled in the state bubble.
In a Mealy Machine, the circuit outputs are a function of both the current state and the circuit inputs. Consequently, the circuit outputs are labeled along the edges of the state diagrams.
We identify the reset state by labeling the state with =0.

4.4 State Tables

State tables describes the behaviour of a sequential circuit in tabular form. We label the columns as current state, next state and output. Based on the inputs, the next state and output may vary.

Chapter 5

State Analysis and Design

5.1 State Analysis Procedure

- Identify the flip flops used to hold the current state information
- Identify the outputs of the circuit
- Write down the logic equations for the circuit outputs and the flip flop inputs(next state equations)
- Use logic equations to derive a state table which describes the next state and circuit outputs
- Obtain a state diagram from the state table

Refer to the notes for a good example

5.2 State Design Procedure

- Understand the verbal description of the problem
- Create a state diagram/table
- Try to reduce the number of states/flip flops
- Assign binary values to each state
- Select a FF type to store binary values
- Derive simplified output equation
- Derive the next-state/flip flop equations
- Draw a schematic of the resulting circuit

Refer to the notes for a good example

5.3 FF Impact on Design

5.3.1 DFF

Look at the state table and derive equations for the next state given the current state and the circuit inputs. Apply the next state logic directly to the DFF inputs

5.3.2 TFF and JKFF

1. Look at stable table to see how the current state changes to the next state for a given current state and inputs
2. Determine how FF input must be set to get desired change from the current state to the next
3. Derive logic equations for the FF inputs given the current state and inputs and apply these connections directly to FF inputs

IMPORTANT: The logic applied to the FF inputs are NOT the next state equations but will generate the correct next state in conjunction with the FF behaviour

5.4 Excitation Tables

Changes in flip flop outputs depending on flip flop inputs is done via an **excitation table**

DFF

D	$Q(t)$	$Q(t+1)$
0	0	1
1	0	1
0	1	0
1	1	1

TFF

D	$Q(t)$	$Q(t+1)$
0	0	0
1	0	1
0	1	1
1	1	0

JKFF

J	K	$Q(t)$	$Q(t+1)$
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	1
1	0	1	1
0	1	1	0
1	1	1	0

5.5 State Assignment

Assigning binary patterns to the symbolic states is known as the state assignment problem.

5.5.1 Encoding

Let's use an example for this:
Assume we have a circuit with 7 states and 3 inputs and 4 outputs. With n bits, we can have 2^n different states. As a result, we'd need 3 flip flops for this example.

However, we can encode the states such that the outputs of the flip-flops are also the output of the circuit(ex: counters). Let's assume that for each state, the outputs are unique(Note: if there were identical outputs, we can add additional bits to distinguish those states). We would use 4 flip flops so that the outputs are the output of the flip flops.

If there were 2 identical outputs, we'd need a 5th flip flop to distinguish both states. The benefit of having more flip flops is that we won't need any output equations.

Therefore, output encoding uses more than the minimum number of flip flops to avoid any output equations.

5.6 State Reduction

Sometimes, we can get more states than required. It's reasonable to attempt to simplify the diagram by removing redundant states. This is the case for states that are equivalent to each other and can be combined.
Two states are **equivalent** if:

- For each input, the state gives the exact same outputs
- For each circuit input, the states give the same next state or an equivalent next state

There are two different methods to reduce:

- Implication charts and merger diagrams
- Partitioning

5.6.1 Implication Charts and Merger Diagrams

Given a state table containing current state, next state and outputs, we can tabulate equivalencies in an implication chart. It resembles the lower left side of a matrix where the rows and columns are the states(1 to n for rows, 0 to $n-1$ for columns) and each entry tells us under which condition two states are equivalent.

1. If two states have different outputs, we set the entry as X
2. Mark entries that are definitely equivalent and mark entries that are equivalent under implied decisions.
3. We perform passes iteratively over the entries from top left to bottom right trying to cross out states that cannot be equivalent due to implied decisions

Based off of the completed implication chart, we can build a a merger diagram i.e. a graph that show merges.
We need to make sure that each state is included and the implied decisions hold.

We can then draw the state table for the new reduced states.

5.6.2 Partitioning

We can divide the states into partitions of equivalent states.
Procedure:

1. Group states according to circuit outputs produced
2. For each group, consider each input pattern. If for any input pattern, different states in a group result in a transition to a different group, then those states are not equivalent so we separate them into 2 smaller groups.
3. This results in fewer states and a smaller state diagram/table

5.7 Algorithmic State Machine

An alternative to a state diagram to implement an algorithm. It is closely tied with hardware implementation and consists of 3 elements:

- **State Box**: Equivalent to state bubble. Has name and binary encoding and Moore Machine outputs(rectangle)
- **Decision Box**: Represents a choice that depends on the inputs(diamond)
- **Conditional Output Box**: Allows specifying outputs that depends both the current state and the inputs, i.e. Mealy outputs (circular)

Note: No output means 0.

5.7.1 Design Steps

We generate a state table by:

1. Determining the number of states from the number of ASM state boxes
2. State assignment for the ASM state boxes
3. Use ASM conditional output boxes, state boxes and decision boxes to determine output values
4. Use decision boxes to determine the next state from the current state

5.7.2 One-Hot Encoding

If we use DFFs and One-Hot Encoding, we can go directly from a ASM diagram to a circuit.

- A state box(Moore Machine) would become a DFF
- A decision box consists of 2 AND gates and 1 inverter along with the 2 inputs: the input data and the decision value

- A conditional output box would tap off the circuitry to generate its output
- Joined edges become an OR gate

The ASM PDF provided gives detailed explanation and examples which explain these concepts better.

5.8 Asynchronous Sequential Circuits

A type of circuit without clocks(therefore NO flip flops) but with the concept of memory using latches and combinational circuits. Due to the absence of clocks, there's a lot of delay due to gate delays.

We may identify asynchronous circuits by the presence of latches and combinational feedback paths(output is fed back into input).
Analysis involves obtaining a table/diagram that describes the sequence of internal states and outputs as a function of changes in circuit inputs.
We will try to obtain transition tables and flow tables to explain asynchronous circuits.

5.8.1 Definitions

Stability: For a given set of inputs, the system is **stable** if the circuit eventually reaches steady state and the excitation variables and secondary variables are equal and unchanging, otherwise, the circuit is unstable.

Fundamental Mode Operation: A circuit is operating in fundamental mode if we assume/force the following restrictions on how the inputs can change:
Primitive flow table: Flow table with only one stable state per row is called a primitive flow table.

- Only one input is allowed to change at a time
- The input changes only after the circuit is stable

5.8.2 Analysis

1. Write logic equations for the excitation variables in terms of the circuit inputs and secondary variables
2. Write logic equations for circuit outputs in terms of the circuit inputs and secondary variables
3. **Transition Table:** Using these equations, we can write a transition table that shows excitation variables and outputs as a function of inputs and secondary variables. 3 Columns: Current state, next state and output
4. **Flow Table**: A transition table with binary numbers replaced with symbols(e.g. a = 00, b = 01 etc.) Same 3 columns as Transition Table
5. Circle stable states where excitation variables are equal to secondary variables

When analyzing circuits with latches that are present, we follow these procedures to analyze the latch:

1. Label each output with Y_i and its feedback path as y_i
2. Derive logic equations for latch inputs S_i and R_j
3. Check if SR=0 for NOR latches and S'R'=0 for NAND latches. If not satisfied, the circuit may not work properly
4. Create logic equations for latch outputs Y_i using known behavior of a latch(Y=S+R'y for NOR Latches and Y=S'+Ry for NAND latches)
5. Construct a transition table using the logic equations for the latch outputs and circuit stable states
6. Obtain a flow table if desired

5.8.3 Design

Very similar to synchronous circuit design:

1. Obtain a primitive flow table from problem description
2. Reduce the flow table to get a smaller flow table with less states
3. Perform state assignment to obtain transition table
4. Obtain next state and output equations(need to avoid hazards and glitches)
5. Draw circuit(with or without latches)

5.8.4 Reduction

Very similar to synchronous circuit reduction:

- Lots of don't care outputs for unstable states
- Don't care next state information if we assume fundamental mode operation

With don't cares, equivalency is replaced with compatibility. Two states A and B are compatible if for every input combination we find:

- A and B produce the same outputs
- A and B have compatible next states where specified

Just like synchronous circuits, we build implication charts, associate compatible states together and build a merger diagram.
It's important to check that each state is included at least once and that any implied compatibilities are true

5.9 Races

5.9.1 Race Condition

A **race condition** occurs in an **asynchronous** when 2 or more state variables change in response to a change in the value of a circuit input.
Unequal circuit delays may imply that the 2 or more state variables may not change simultaneously - this may cause a problem.

Assume 2 state variables change:

- If the circuit reaches the same final, stable state regardless of the order the state variables change, then the race is **non-critical**
- If the circuit reaches a different final, stable state depending on the order in which the state variables change, then the race is **critical**

We need to avoid critical races for predictability and to ensure our circuit does the intended function

5.9.2 Stability

Assume we start at a 00 state. We need to be careful that our circuit does not become unstable, meaning its state oscillates from a few different states without stabilizing onto one.

5.9.3 Race Free State Assignment

We can prevent races by performing state assignments such that transitions from one stable state to another require only 1 state variable to change at a time Techniques include:

- Build a transition diagram based on a flow table where each is a transition and each node is a state. Embed the symbolic states into the coordinates of a n dimensional cube such that the path from stable state to another is:
  - Along a single edge of a cube
  - Through along edges of the cube
- Replace a state with multiple equivalent states with equivalent outputs. That way, when building a transition diagram, there'll always be 1 of the 2 equivalent states directly adjacent to every other state
- Using the idea of one-hot encoding, given n states, let 00..1..0 where 1 is the ith state. Each transition to a different state, we introduce an unstable state from i to j such that 00..1_i...1_j..00

5.10 Hazard and Output Glitches

5.10.1 Definition

A **hazard** is a momentary unwanted switching transient at a logic function's output due to unequal propagation delays along different paths in a combinational circuit.

There are 2 types of hazards:**static** and **dynamic**
For asynchronous circuits in particular, hazards can cause problems in addition to other issues like races and non-fundamental mode operation. Momentary false logic function values in an asynchronous circuit can cause a transition to an incorrect stable state.

5.10.2 Static Hazards

Static-0: Occurs when the output should remain 0 but temporarily switches to 1 due to a change in input.
Static-1: Occurs when the output should remain 1 ut temporarily switches to 0 due to a change in input.

5.10.3 Dynamic Hazard

Occurs when an input changes and a circuit output should change but temporarily flips between values. Ex: 0-¿1-¿0-¿1 or 1-¿0-¿1-¿0

5.10.4 Fixing Hazards

When circuits are implemented as 2 level SOP, we can detect hazards by inspecting K-Maps and adding redundant product terms. As an input changes, there's a possibility that we jump from one product term to another(one rectangle to another). If adjacent minterms are not covered by the same product term, then a **hazard exists**. To fix this, we add a redundant product term connecting both product terms.

Fixing this guarantees that there will be no static or dynamic hazards. The same can be applied with POS by adding redundant sum terms

5.10.5 Hazards in Asynchronous Circuits

Draw a timing diagram to visualize the delay. To draw it, make vertical slices of width Δt where Δt is the delay for each gate and then trace through the whole circuit based on the input switch.

To fix it, we can use latches. If we're trying to fix a static-1 hazard, we need to be able to tolerate momentary 0 $\Leftarrow$ Use a **SR Latch(NOR Latch)**
We can then draw the K-Maps for both S and R inputs for the circuit.
This is because:

- An SR(NOR) latch can tolerate momentary 0s appearing at its input(since we momentarily move from a set or reset to a hold and then back).
- An S'R'(NAND) latch can tolerate momentary 1s appearing at its input(since we momentarily move from a set or reset to a hold and then back).

5.10.6 Output Assignment in Asynchronous Circuits

If a flow/transition table has unspecified entries for circuit outputs, it may be due to the fundamental mode assumption or unstable states(stable states always have specified outputs).

Unspecified outputs can lead to don't care which helps during minimization of logic equations(we may pass through these values from state to state).
Our circuit may have glitches due to our derived equations however.

To avoid output glitches, consider transitions between two states:

- If both stable states produce a 0 output, make output 0 instead of a don't care.
- If both stable states produce a 1 output, make output 1 instead of a don't care.
- If stable states produce different outputs, the output can remain a don't care and be used to find a smaller output circuit.