ELECTRICAL
Engineering KMITNB

Power | Control | Communication | Computer

Digital Circuit and Logic Design

Lecture 8-3 : Synchronous Sequential Circuits

- Implementations Synchronous Sequential Circuits using D-type, T-type and JK-type
- VHDL for Sequential Circuits
- State Reduction

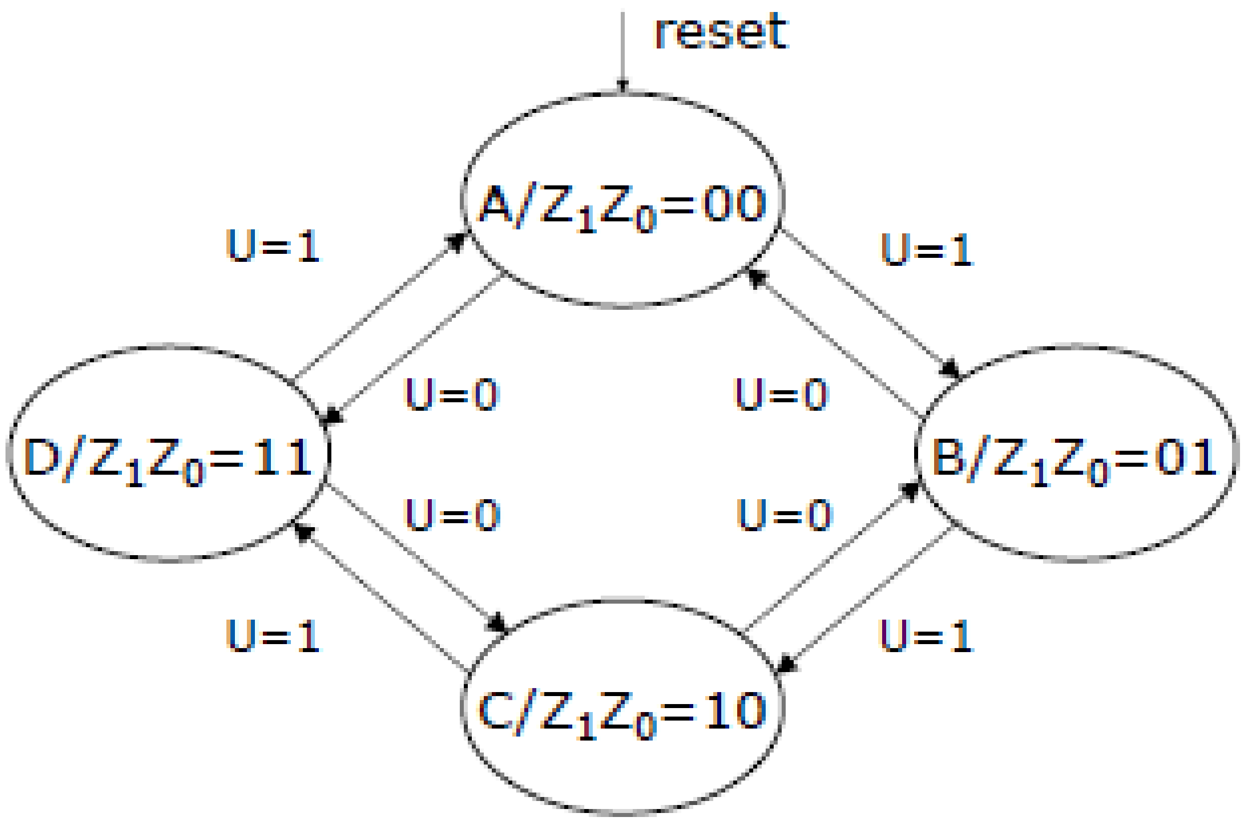
Objectives

- Implementations Synchronous Sequential Circuits using D-type, T-type and JK-type
- VHDL for Sequential Circuits
- State Reduction

Counter design example

- **Design a 2-bit counter that counts**
  - in the sequence 0,1,2,3,0,… if a given control signal U=1, or
  - in the sequence 0,3,2,1,0,… if a given control signal U=0
- This represents a 2-bit binary up/down counter
  - An input U to control to count direction
  - A RESET input to reset the counter to the value zero
  - Two outputs (Z1Z0) representing the output (0-3)
  - Counter counts on positive edge transitions of a common clock signal
- Design this counter as a synchronous sequential machine using
- **D-type, T-type, JK-type flip-flops**

Counter state diagram



Counter state table

Present state	Next state		Output Z_1Z_0
	U=0	U=1	
A	D	B	00
B	A	C	01
C	B	D	10
D	C	A	11

State-assigned state table

Choosing a state assignment of A=00, B=01, C=10 and D=11 makes sense here because the outputs Z1Z0 become the outputs from the flip-flops directly

	Present state Y_2Y_1	Next state		Output Z_1Z_0
		U=0	U=1	
		Y_2Y_1	Y_2Y_1	
A	00	11	01	00
B	01	00	10	01
C	10	01	11	10
D	11	10	00	11

D-type flip-flop implementation

- When D flip-flops are used to implement an FSM, the next-state entries in the state assigned state table correspond directly to the signals that must be applied to the D inputs
- Thus, K-maps for the D inputs can be derived directly from the state-assigned state table
- This will not be the case for the other types of flip-flops (T, JK)

State table and next-state maps

	Present state y_2y_1	Next state $U=0$ Y_2Y_1	Next state $U=1$ Y_2Y_1	Output Z_1Z_0
A	00	11	01	00
B	01	00	10	01
C	10	01	11	10
D	11	10	00	11

$Z_1=y_2$ $\quad Z_0=y_1$

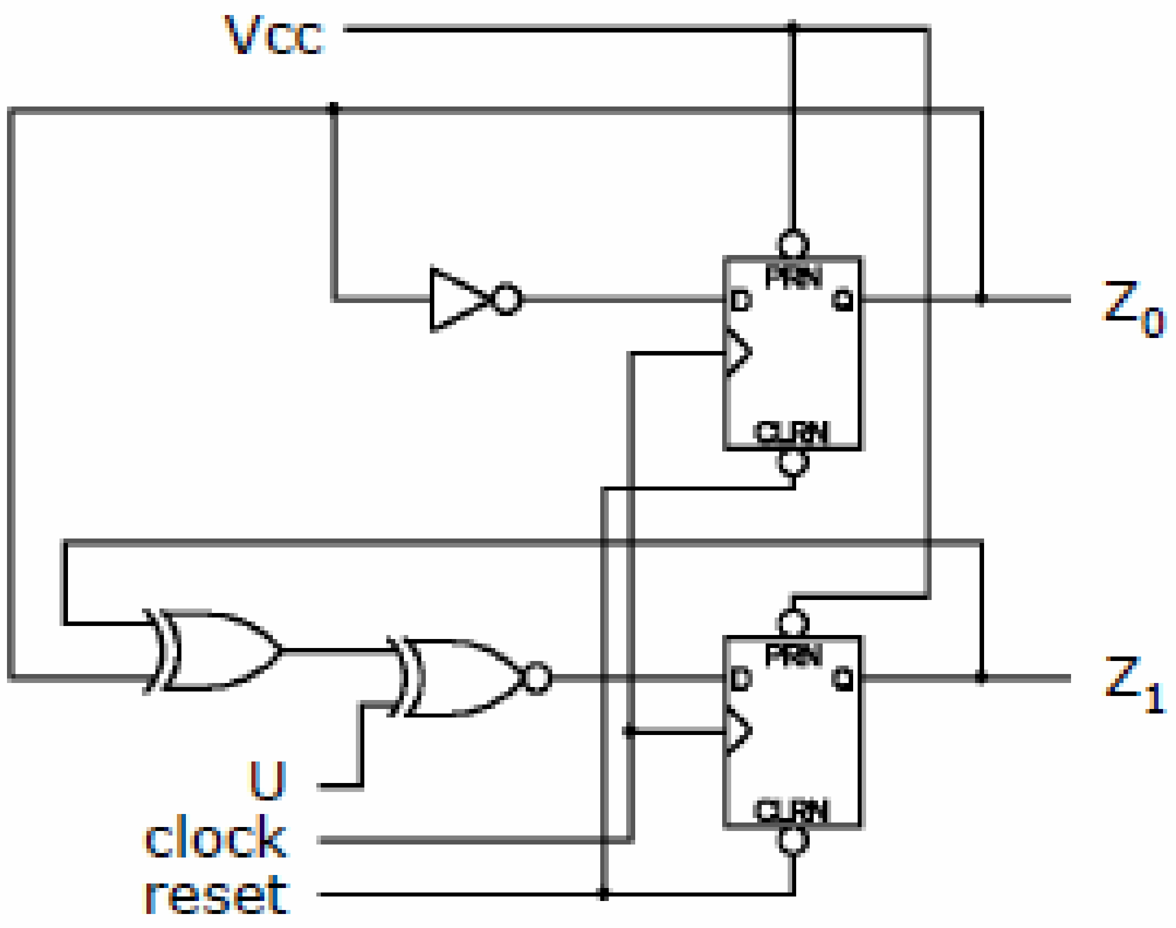
u \ y_2y_1	00	01	11	10
0	1	0	0	1
1	1	0	0	1

$Y_1=y_1'$

u \ y_2y_1	00	01	11	10
0	1	0	1	0
1	0	1	0	1

$Y_2=(y_2 \oplus y_1 \oplus u)'$

Circuit diagram (D flip-flop)



Design using other flip-flop types

- For the T- or JK-type flip-flops, we must derive the desired inputs to the flip-flops
- Begin by constructing a **transition table** for the flip-flop type you wish to use
  - This table simply lists required inputs for a given change of state
- The transition table is used with the state assigned state table to construct an **excitation table**
  - The excitation table lists the required flip-flop inputs that must be 'excited' to cause a transition to the next state

Transition tables

J	K	Q	Q^+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Q	Q^+	J	K
0	0	0	D
0	1	1	D
1	0	D	1
1	1	D	0

JK transition table

T	Q	Q^+
0	0	0
0	1	1
1	0	1
1	1	0

Q	Q^+	T
0	0	0
0	1	1
1	0	1
1	1	0

T transition table

The transition table lists required flip-flop inputs to affect a specific change

T-type flip-flop implementation

Use entries from the transition table to derive the flip-flop inputs based on the state-assigned state table.

Q	Q^+	T
0	0	0
0	1	1
1	0	1
1	1	0

excitation table

Present state y_2y_1	Flip-flop inputs U=0		U=1		Output z_1z_0
	Y_2Y_1	T_2T_1	Y_2Y_1	T_2T_1	
00	11	11	01	01	00
01	00	01	10	11	01
10	01	11	11	01	10
11	10	01	00	11	11

Excitation table and K-maps

Present state y_2y_1	Flip-flop inputs		Output Z_1Z_0
	U=0	U=1	
	T_2T_1	T_2T_1	
00	11	01	00
01	01	11	01
10	11	01	10
11	01	11	11

$Z_1=y_2 \quad Z_0=y_1$

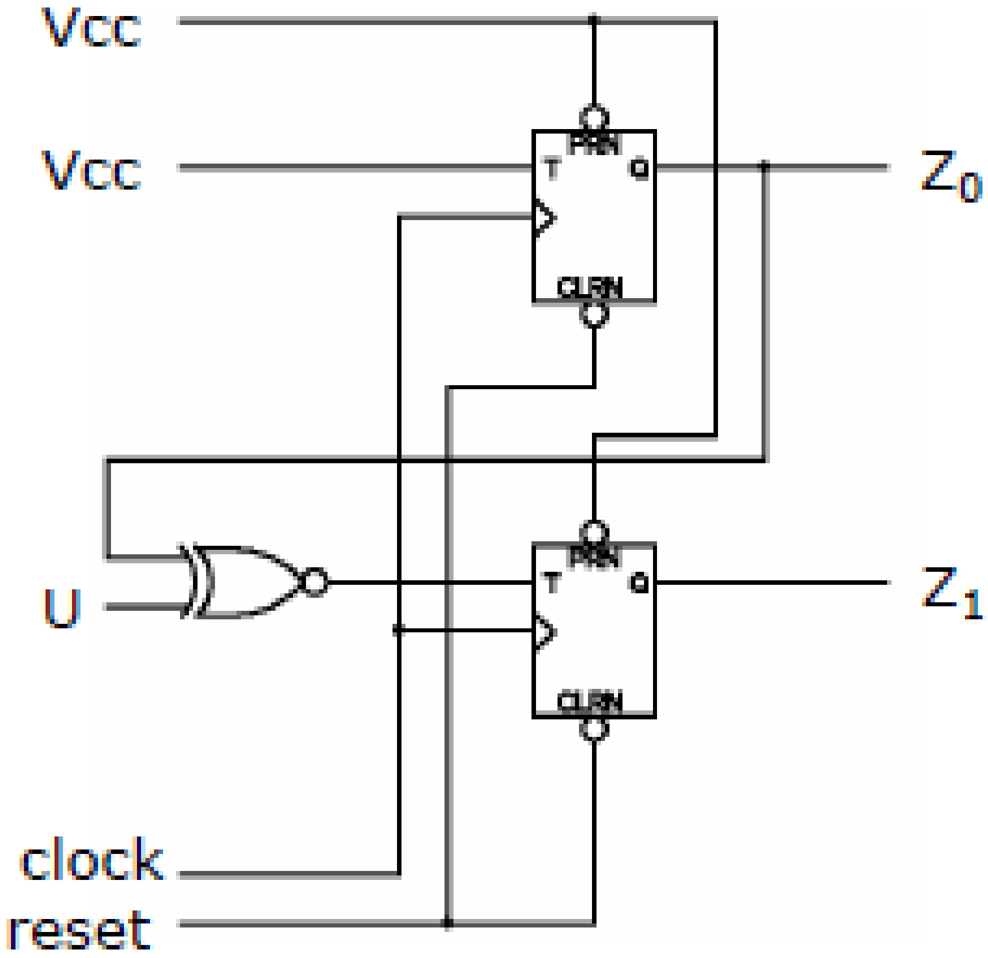
$u \backslash y_2y_1$	00	01	11	10
0	1	1	1	1
1	1	1	1	1

$T_1=1$

$u \backslash y_2y_1$	00	01	11	10
0	1	0	0	1
1	0	1	1	0

$T_2=y_1u+y_1'u'=(y_1\oplus u)'$

Circuit diagram (T flip-flop)



JK-type flip-flop implementation

- Use entries from the transition table to derive the flip-flop inputs based on the state-assigned state table
  - This must be done for each input (J and K) on each flip-flop

Present state Y_2Y_1	Next state U=0 Y_2Y_1	Next state U=1 Y_2Y_1	Output Z_1Z_0
00	11	01	00
01	00	10	01
10	01	11	10
11	10	00	11

Q	Q^+	J	K
0	0	0	D
0	1	1	D
1	0	D	1
1	1	D	0

JK transition table

JK-type flip-flop implementation

Q	Q^+	J	K
0	0	0	D
0	1	1	D
1	0	D	1
1	1	D	0

JK transition table

Present state y_2y_1	Flip-flop inputs U=0			Flip-flop inputs U=1			Output Z_1Z_0
	Y_2Y_1	J_2K_2	J_1K_1	Y_2Y_1	J_2K_2	J_1K_1	
00	11	1D	1D	01	0D	1D	00
01	00	0D	D1	10	1D	D1	01
10	01	D1	1D	11	D0	1D	10
11	10	D0	D1	00	D1	D1	11

Excitation table and K-maps

Present state y_2y_1	Flip-flop inputs						Output Z_1Z_0
	U=0			U=1			
	Y_2Y_1	J_2K_2	J_1K_1	Y_2Y_1	J_2K_2	J_1K_1	
00	11	1D	1D	01	0D	1D	00
01	00	0D	D1	10	1D	D1	01
10	01	D1	1D	11	D0	1D	10
11	10	D0	D1	00	D1	D1	11

u \ y_2y_1	00	01	11	10
0	1	D	D	1
1	1	D	D	1

$J_1=1$

u \ y_2y_1	00	01	11	10
0	D	1	1	D
1	D	1	1	D

$K_1=1$

Excitation table and K-maps

Present state y_2y_1	Flip-flop inputs						Output Z_1Z_0
	U=0			U=1			
	Y_2Y_1	J_2K_2	J_1K_1	Y_2Y_1	J_2K_2	J_1K_1	
00	11	1D	1D	01	0D	1D	00
01	00	0D	D1	10	1D	D1	01
10	01	D1	1D	11	D0	1D	10
11	10	D0	D1	00	D1	D1	11

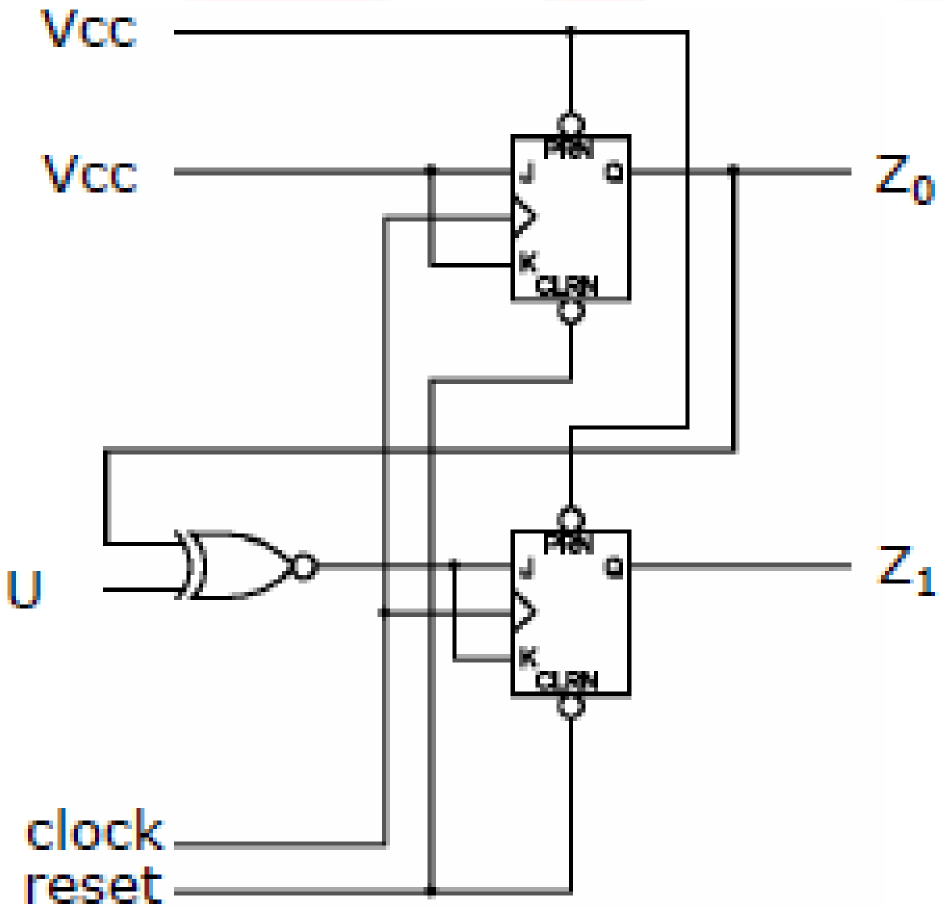
$u \backslash y_2y_1$	00	01	11	10
0	1	0	D	D
1	0	1	D	D

$J_2 = (y_1 \oplus u)'$

$u \backslash y_2y_1$	00	01	11	10
0	D	D	0	1
1	D	D	1	0

$K_2 = (y_1 \oplus u)'$

Circuit diagram (JK flip-flop)



Analysis with JK flip-flops

- Determine the flip-flop input function in terms of the present state and input variables
- Used the corresponding flip-flop characteristic table to determine the next state

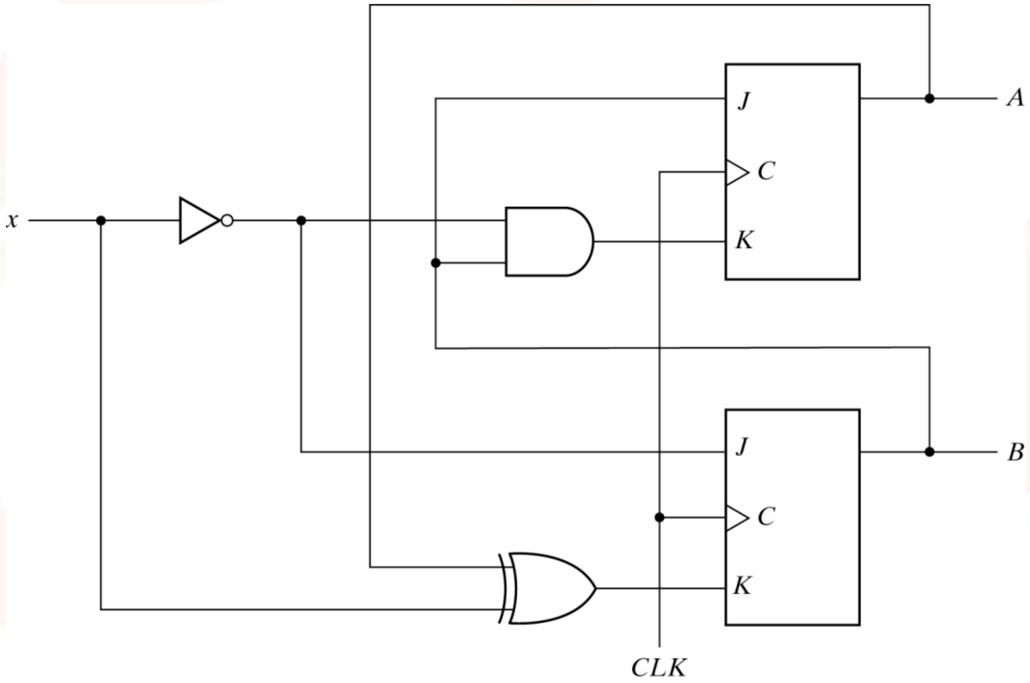


Fig. 5-18 Sequential Circuit with *JK* Flip-Flop

– $JA = B$, $KA = Bx'$
– $JB = x'$, $KB = A'x + Ax'$
– derive the state table

Present state		Input	Next state		Flip-flop inputs			
A	*B*	*x*	*A*	*B*	*JA*	*KA*	*JB*	*KB*
0	0	0	0	1	0	0	1	0
0	0	1	0	0	0	0	0	1
0	1	0	1	1	1	1	1	0
0	1	1	1	0	1	0	0	1
1	0	0	1	1	0	0	1	1
1	0	1	1	0	0	0	0	0
1	1	0	0	0	1	1	1	1
1	1	1	1	1	1	0	0	0

– Or, derive the state equations using characteristic equation.

State transition diagram

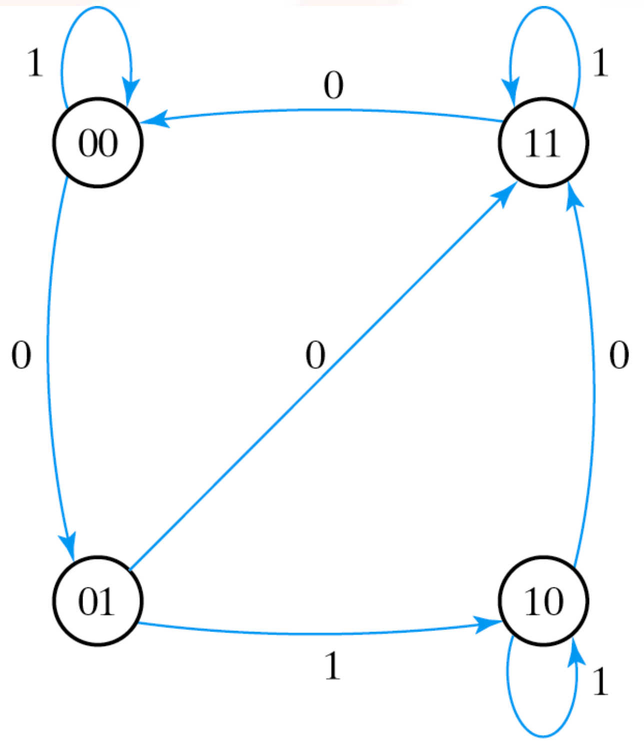


Fig. 5-19 State Diagram of the Circuit of Fig. 5-18

VHDL for Sequential Circuits

- D Latches
- D Flip-Flop
- Shift Register
- Counter

Using a D flip-flop package

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
LIBRARY altera ;
USE altera.maxplus2.all ;

ENTITY flipflop IS
    PORT ( D, Clock          : IN    STD_LOGIC ;
           Resetn, Presetn   : IN    STD_LOGIC ;
           Q                 : OUT   STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Structure OF flipflop IS
BEGIN
    dff_instance: dff PORT MAP ( D, Clock, Resetn, Presetn, Q ) ;
END Structure ;
```

Active low signals

Code for a gated D latch

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY latch IS
   PORT (     D, Clk          : IN    STD_LOGIC ;
              Q               : OUT STD_LOGIC) ;
END latch ;

ARCHITECTURE Behavior OF latch IS
BEGIN
   PROCESS ( D, Clk )
   BEGIN
        IF Clk = '1' THEN
              Q <= D ;
        END IF ;
   END PROCESS ;
END Behavior ;
```

USES IMPLIED MEMORY

Code for a D flip-flop

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
   PORT (     D, Clock        : IN    STD_LOGIC ;
              Q               : OUT STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
   PROCESS ( Clock )
   BEGIN
       IF Clock'EVENT AND Clock = '1' THEN
              Q <= D ;
       END IF ;
   END PROCESS ;
END Behavior ;
```

POSITIVE EDGE TRIGGERED

Code for a D flip-flop (alternate)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY flipflop IS
    PORT (    D, Clock      : IN   STD_LOGIC ;
              Q             : OUT  STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
   PROCESS
   BEGIN
      WAIT UNTIL Clock'EVENT AND Clock = '1' ;
      Q <= D ;
   END PROCESS ;
END Behavior ;
```

POSITIVE EDGE TRIGGERED

D flip-flop with synchronous reset

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
   PORT (     D, Resetn, Clock        : IN     STD_LOGIC ;
              Q                       : OUT    STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
   PROCESS
   BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF Resetn = '0' THEN
              Q <= '0' ;
        ELSE
              Q <= D ;
        END IF ;
   END PROCESS ;
END Behavior ;
```

D flip-flop with MUX input

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY muxdff IS
   PORT (     D0, D1, Sel, Clock     : IN    STD_LOGIC ;
              Q                      : OUT   STD_LOGIC ) ;
END muxdff ;

ARCHITECTURE Behavior OF muxdff IS
BEGIN
   PROCESS
   BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF Sel = '0' THEN
              Q <= D0 ;
        ELSE
              Q <= D1 ;
        END IF ;
   END PROCESS ;
END Behavior ;
```

Four-bit shift register

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY shift4 IS
   PORT (      R : IN           STD_LOGIC_VECTOR(3 DOWNTO 0) ;
               L, w, Clock      : IN     STD_LOGIC ;
               Q : BUFFER       STD_LOGIC_VECTOR(3 DOWNTO 0) )
   ;
END shift4 ;

ARCHITECTURE Structure OF shift4 IS
   COMPONENT muxdff
      PORT (        D0, D1, Sel, Clock     : IN    STD_LOGIC ;
                    Q                      : OUT   STD_LOGIC ) ;
   END COMPONENT ;
BEGIN
   Stage3: muxdff PORT MAP ( w, R(3), L, Clock, Q(3) ) ;
   Stage2: muxdff PORT MAP ( Q(3), R(2), L, Clock, Q(2) ) ;
   Stage1: muxdff PORT MAP ( Q(2), R(1), L, Clock, Q(1) ) ;
   Stage0: muxdff PORT MAP ( Q(1), R(0), L, Clock, Q(0) ) ;
END Structure ;
```

Alternate code for shift register

```
ENTITY shift4 IS
   PORT (      R        : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
               Clock    : IN     STD_LOGIC ;
               L, w     : IN     STD_LOGIC ;
               Q        : BUFFER  STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END shift4 ;

ARCHITECTURE Behavior OF shift4 IS
BEGIN
   PROCESS
   BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF L = '1' THEN
               Q <= R ;
        ELSE
               Q(0) <= Q(1) ;
               Q(1) <= Q(2);
               Q(2) <= Q(3) ;
               Q(3) <= w ;
        END IF ;
   END PROCESS ;
END Behavior ;
```

Four-bit up counter

```
ARCHITECTURE Behavior OF upcount IS
   SIGNAL Count : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
BEGIN
   PROCESS ( Clock, Resetn )
   BEGIN
        IF Resetn = '0' THEN
              Count <= "0000" ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
              IF E = '1' THEN
                    Count <= Count + 1 ;
              ELSE
                    Count <= Count ;
              END IF ;
        END IF ;
   END PROCESS ;
   Q <= Count ;
END Behavior ;
```

Four-bit up counter with load

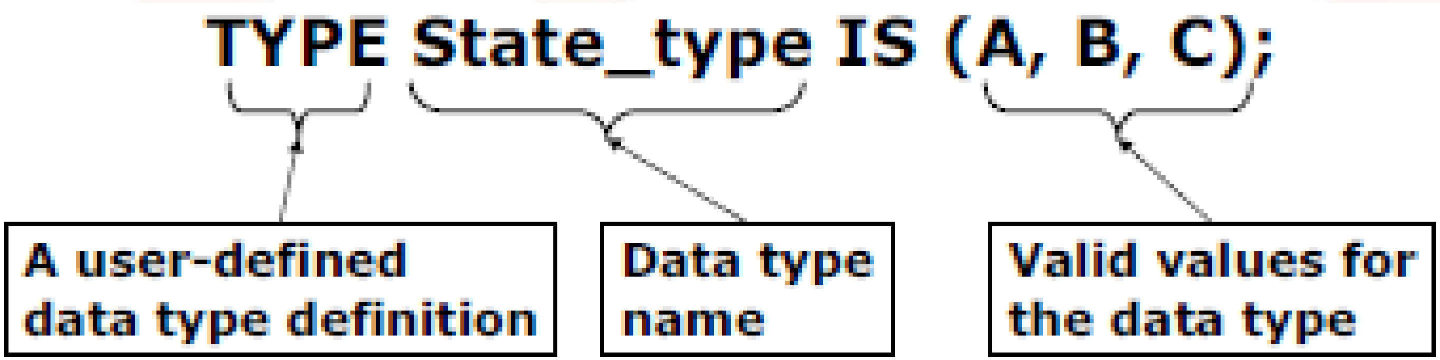
```
ENTITY upcount IS
   PORT (      R          : IN            INTEGER RANGE 0 TO 15 ;
               Clock, Resetn, L          : IN           STD_LOGIC ;
               Q          : BUFFER        INTEGER RANGE 0 TO 15 ) ;
END upcount ;
ARCHITECTURE Behavior OF upcount IS
BEGIN
   PROCESS ( Clock, Resetn )
   BEGIN
        IF Resetn = '0' THEN
               Q <= 0 ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
               IF L = '1' THEN
                      Q <= R ;
               ELSE
                      Q <= Q + 1 ;
               END IF;
        END IF;
   END PROCESS;
END Behavior;
```

FSM design using CAD tools

- VHDL provides a number of constructs for designing finite state machines
- There is not a standard way for defining an FSM
- Basic approach
  - Create a user-defined data type to represent the possible states of an FSM
  - This signal represents the outputs (state variables) of the flip-flops that implement the states in the FSM
  - VHDL compiler chooses the appropriate number of flip-flops during the synthesis process
  - The state assignment can be done by the compiler or can be user specified

User defined data types

- The **TYPE** keyword will be used to define a new data type used to represent states in the FSM



Defines a data type (called State_type) that can take on three distinct values: A, B, or C).

Representing states

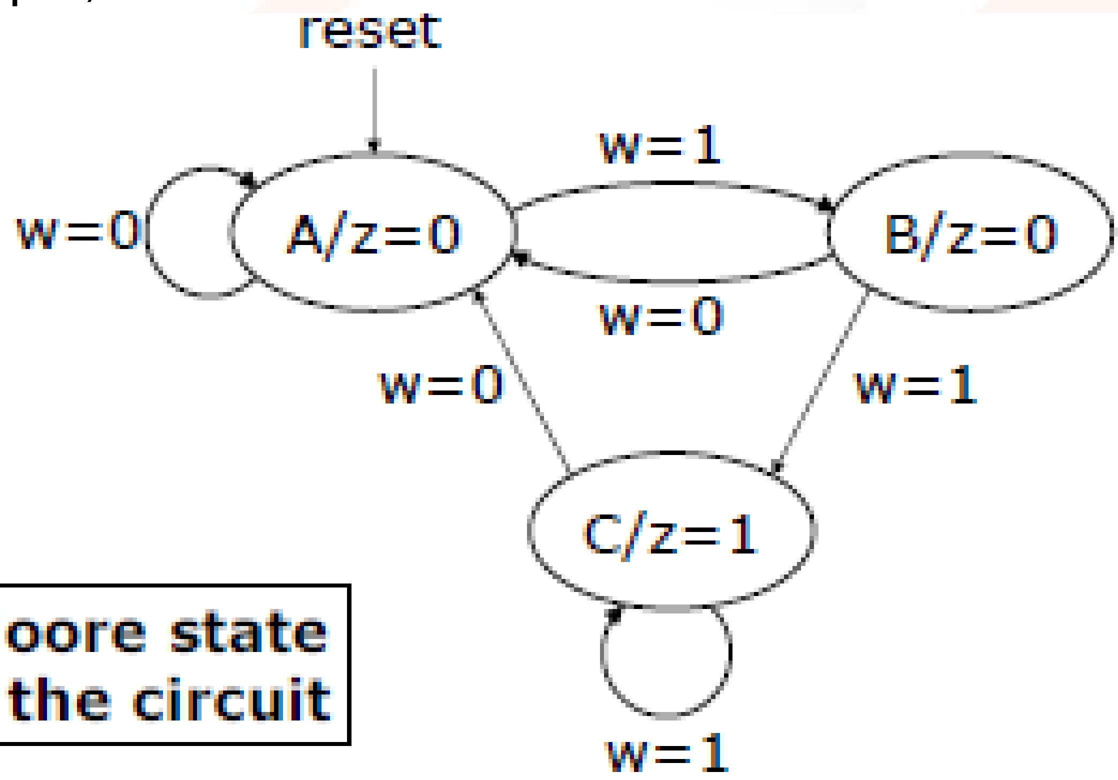
- A **SIGNAL** is defined, of the user-defined **State_type,** to represent the flip-flop outputs

```
TYPE State_type IS (A, B, C);
SIGNAL y: State_type;
```

The signal, y, can be used to represent the flip-flop outputs for an FSM that has three states

Design example

- Create a VHDL description for a circuit that detects a ’11’ input sequence on an input, **w**



Recall the Moore state diagram for the circuit

VHDL design example

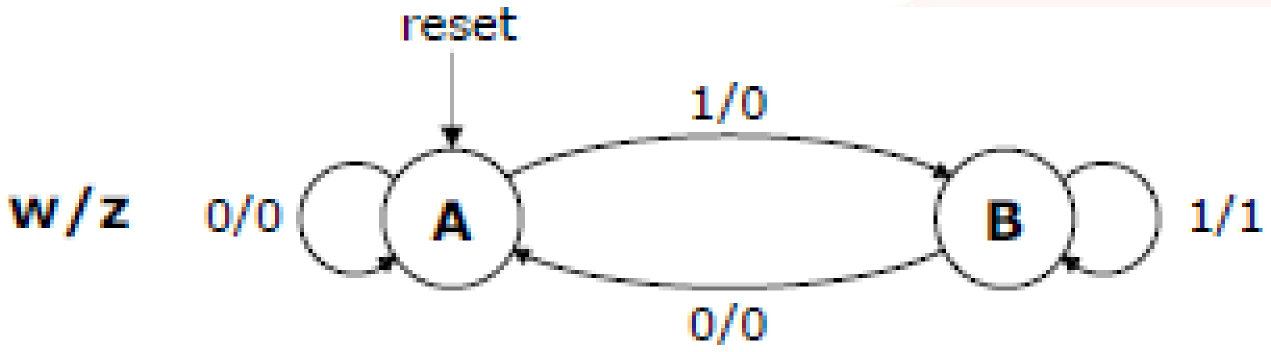
```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY detect IS
   PORT(      clk, resetn, w  : IN    STD_LOGIC ;
              z               : OUT   STD_LOGIC) ;
END detect ;

ARCHITECTURE Behavior OF detect IS
   TYPE State_type IS (A,B,C) ;
   SIGNAL y: State_type ;
BEGIN
```

```
PROCESS ( resetn, clk )
BEGIN
  IF resetn = '0' THEN
    y <= A;
  ELSIF (clk'EVENT AND clk='1') THEN
    CASE y IS
      WHEN A =>
        IF w='0' THEN
          y <= A;
        ELSE
          y <= B;
        END IF;
      WHEN B =>
        IF w='0' THEN
          y <= A;
        ELSE
          y <= C;
        END IF;
      WHEN C =>
        IF w='0' THEN
          y <= A;
        ELSE
          y <= C;
        END IF;
    END CASE;
  END IF;
END PROCESS
z <= '1' WHEN y=C ELSE '0';
END Behavior;
```

VHDL code of a Mealy FSM

- A Mealy FSM can be described in a similar manner as a Moore FSM
- The state transitions are described in the same way as the original VHDL example
- The major difference in the case of a Mealy FSM is the way in which the code for the output is written
- Recall the Mealy state diagram for the ’11’ sequence detector

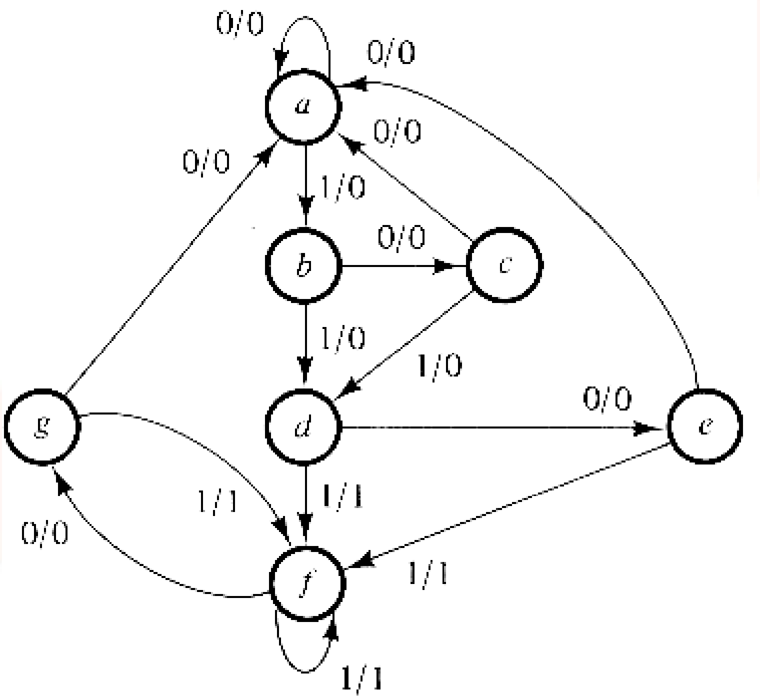


Mealy ’11’ detector VHDL code

```
ARCHITECTURE Behavior OF detect IS
    TYPE State_type IS (A,B) ;
    SIGNAL y: State_type ;
BEGIN
PROCESS(resetn,clk)
BEGIN
    IF resetn='0' THEN
        y <= A;
    ELSEIF(clk'EVENT AND clk='1') THEN
     CASE y IS
        WHEN A =>
          IF w='0' THEN y<= A;
          ELSE y<= B;
          END IF;
        WHEN B =>
          IF w='0' THEN y<= A;
          ELSE y<= B;
          END IF;
     END CASE;
   END IF;
END PROCESS;
PROCESS(y,w)
BEGIN
   CASE y IS
     WHEN A =>
        z <='0';
     WHEN B =>
        z <= w;
   END CASE;
END PROCESS;
END Behavior;
```

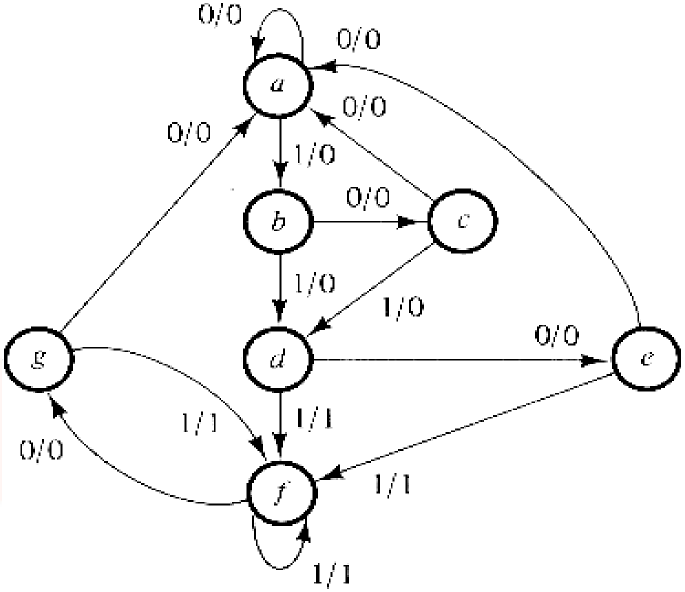
State Reduction

- State Reduction
  - reductions on the number of flip-flops and the number of gates
  - a reduction in the number of states may result in a reduction in the number of flip-flops
  - a example state diagram



State Reduction

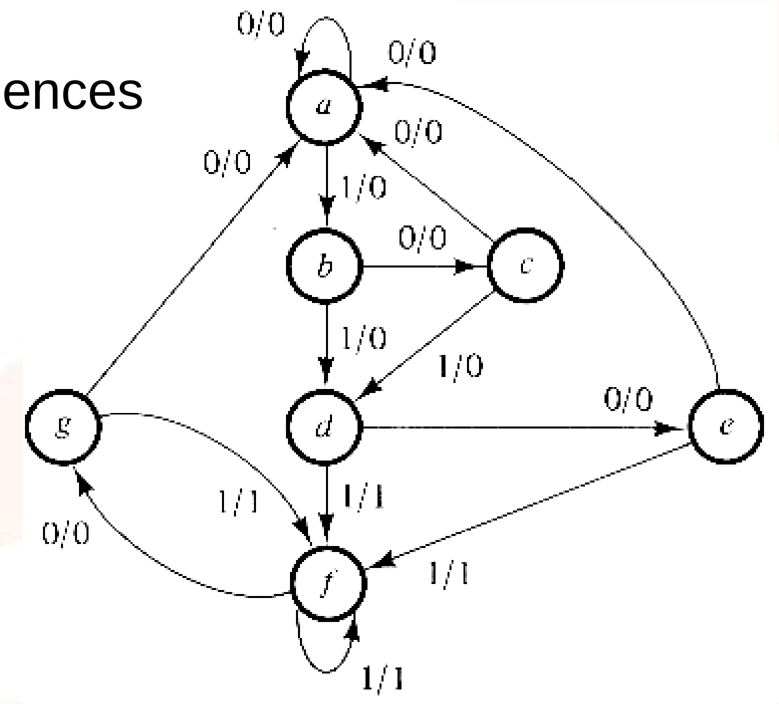
- State Reduction
  - reductions on the number of flip-flops and the number of gates
  - a reduction in the number of states may result in a reduction in the number of flip-flops
  - a example state diagram



– state a a b c d e f f g f g a
input 0 1 0 1 0 1 1 0 1 0 0
output 0 0 0 0 0 1 1 0 1 0 0

– only the input-output sequences are important

– two circuits are equivalent
  - have identical outputs for all input sequences
  - the number of states is not important



Equivalent states

- two states are said to be equivalent
  - for each member of the set of inputs, they give exactly the same output and send the circuit to the same state or to an equivalent state
  - one of them can be removed

Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	*a*	*b*	0	0
b	*c*	*d*	0	0
c	*a*	*d*	0	0
d	*e*	*f*	0	1
e	*a*	*f*	0	1
f	*g*	*f*	0	1
g	*a*	*f*	0	1

ELECTRICAL
Engineering KMITNB

- Reducing the state table
  - e=g
  - d=?

Present State	Next State $x = 0$	Next State $x = 1$	Output $x = 0$	Output $x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	g	f	0	1
g	a	f	0	1

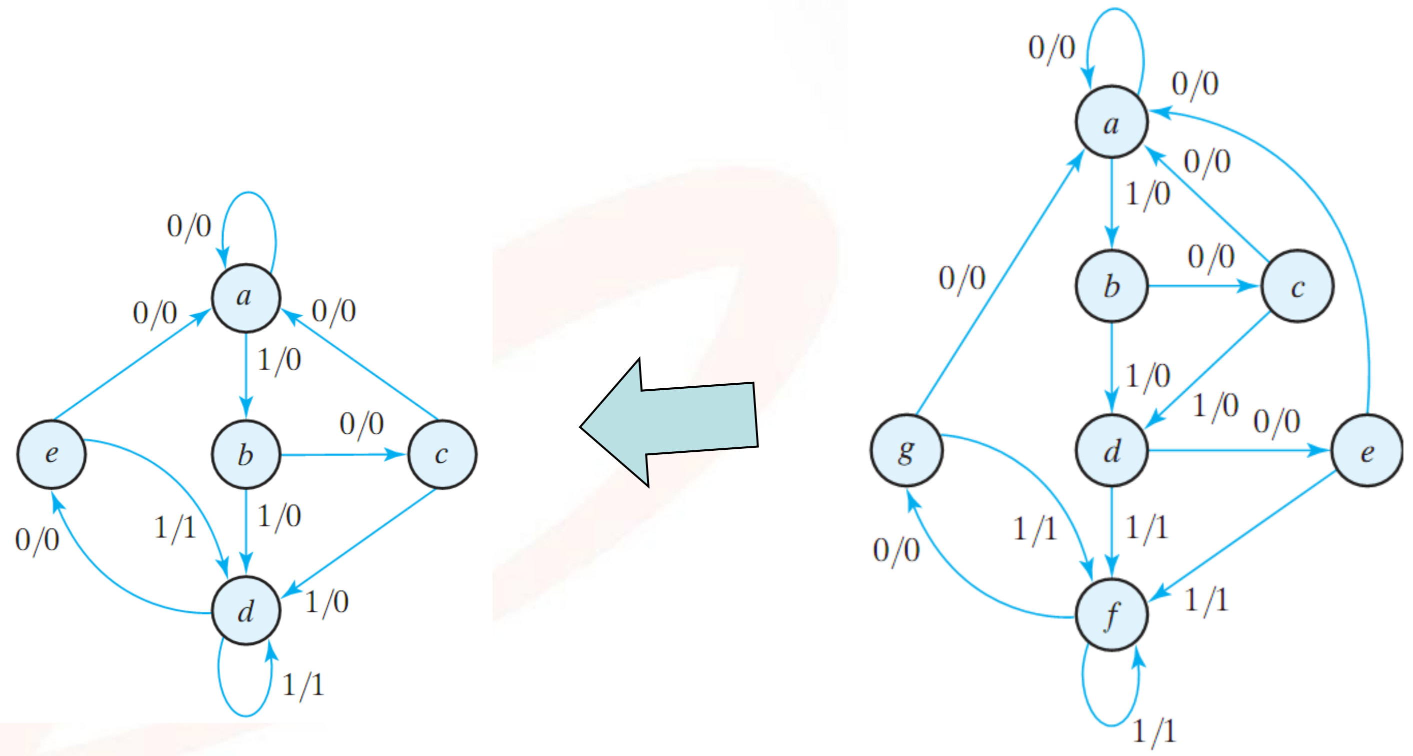
g => e

Present State	Next State $x = 0$	Next State $x = 1$	Output $x = 0$	Output $x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	e	f	0	1

– the reduced finite state machine

Present State	Next state		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	d	0	1
e	a	d	0	1

– state a a b c d e d d e d e a
input 0 1 0 1 0 1 1 0 1 0 0
output 0 0 0 0 0 1 1 0 1 0 0



– the checking of each pair of states for possible equivalence can be done systematically

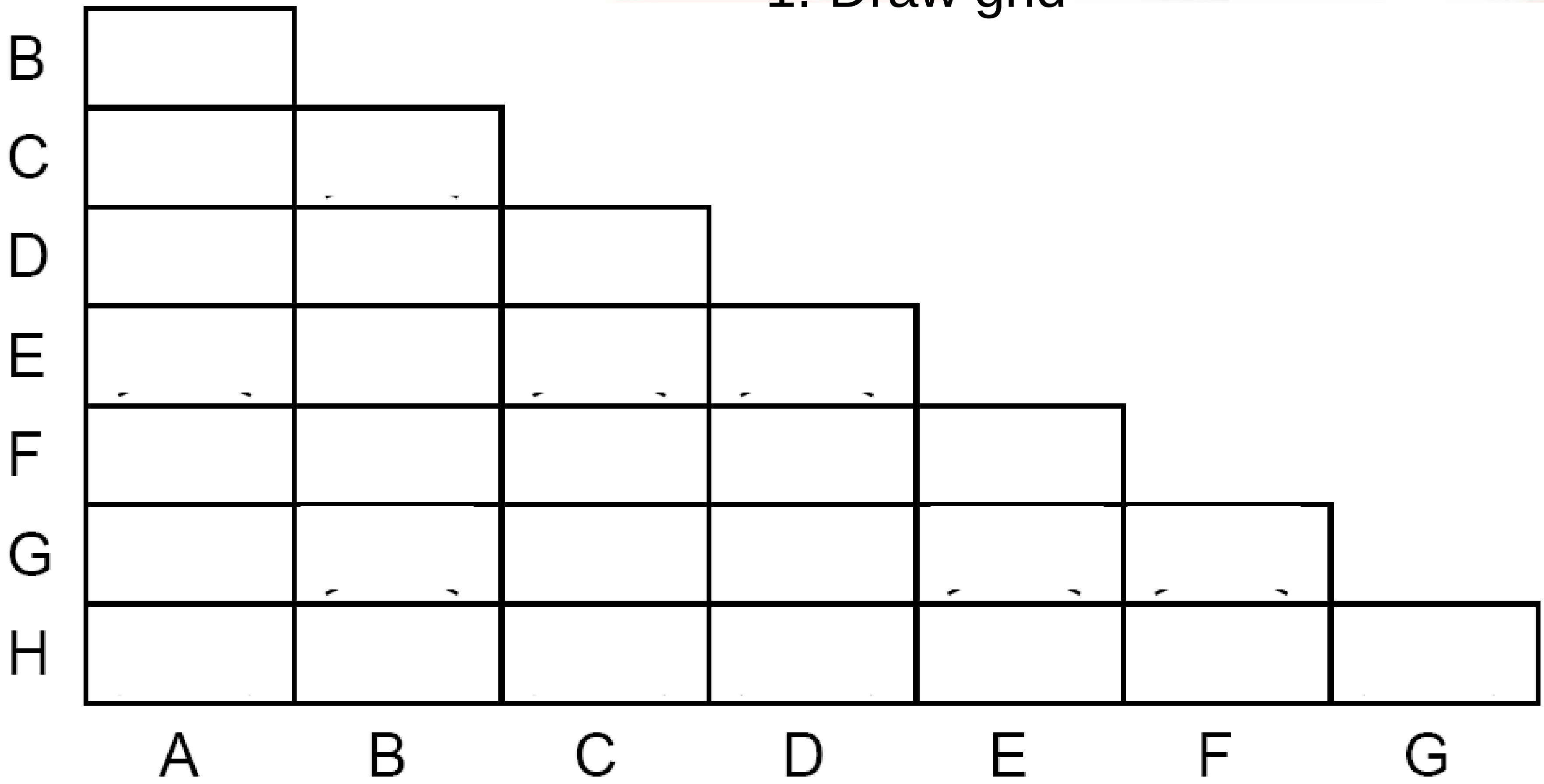
– the unused states are treated as don't-care condition $\Rightarrow$ fewer combinational gates

Implication Charts

Present state	Next state		Output Z	
	X=0	X=1	X=0	X=1
A	G	B	1	0
B	F	A	0	1
C	C	F	1	0
D	G	E	1	0
E	H	G	0	1
F	C	A	0	1
G	D	H	1	0
H	E	D	0	1

Implication Charts

1. Draw grid



Implication Charts

2. Where states do not have the same outputs, place an x

	A	B	C	D	E	F	G
B	X						
C		X					
D		X					
E	X		X	X			
F	X		X	X			
G		X			X	X	
H	X		X	X			X

Implication Charts

3. Where states have the same output, put equalities in the

A	G	B	1	0
C	C	F	1	0

	A	B	C	D	E	F	G
B	×						
C	$C \equiv G$ $B \equiv F$	×					
D	$B \equiv E$	×	$C \equiv G$ $E \equiv F$				
E	×	$F \equiv H$ $A \equiv G$	×	×			
F	×	$C \equiv F$	×	×	$C \equiv H$ $A \equiv G$		
G	$D \equiv G$ $B \equiv H$	×	$C \equiv D$ $F \equiv H$	$E \equiv H$	×	×	
H	×	$E \equiv F$ $A \equiv D$	×	×	$D \equiv G$	$C \equiv E$ $A \equiv D$	×

Implication Charts

B	X						
C	$C \equiv G$ $B \equiv F$	X					
D	$B \equiv E$	X	$C \equiv G$ $E \equiv F$				
E	X	$F \equiv H$ $A \equiv G$	X	X			
F	X	$\cancel{C \equiv F}$	X	X	$C \equiv H$ $A \equiv G$		
G	$D \equiv G$ $B \equiv H$	X	$C \equiv D$ $F \equiv H$	$E \equiv H$	X	X	
H	X	$E \equiv F$ $A \equiv D$	X	X	$D \equiv G$	$C \equiv E$ $A \equiv D$	X
	A	B	C	D	E	F	G

4. Where equivalents are false, cross out the box

(e.g. B,F has C≡F as a false equivalent as C,F has no contents)

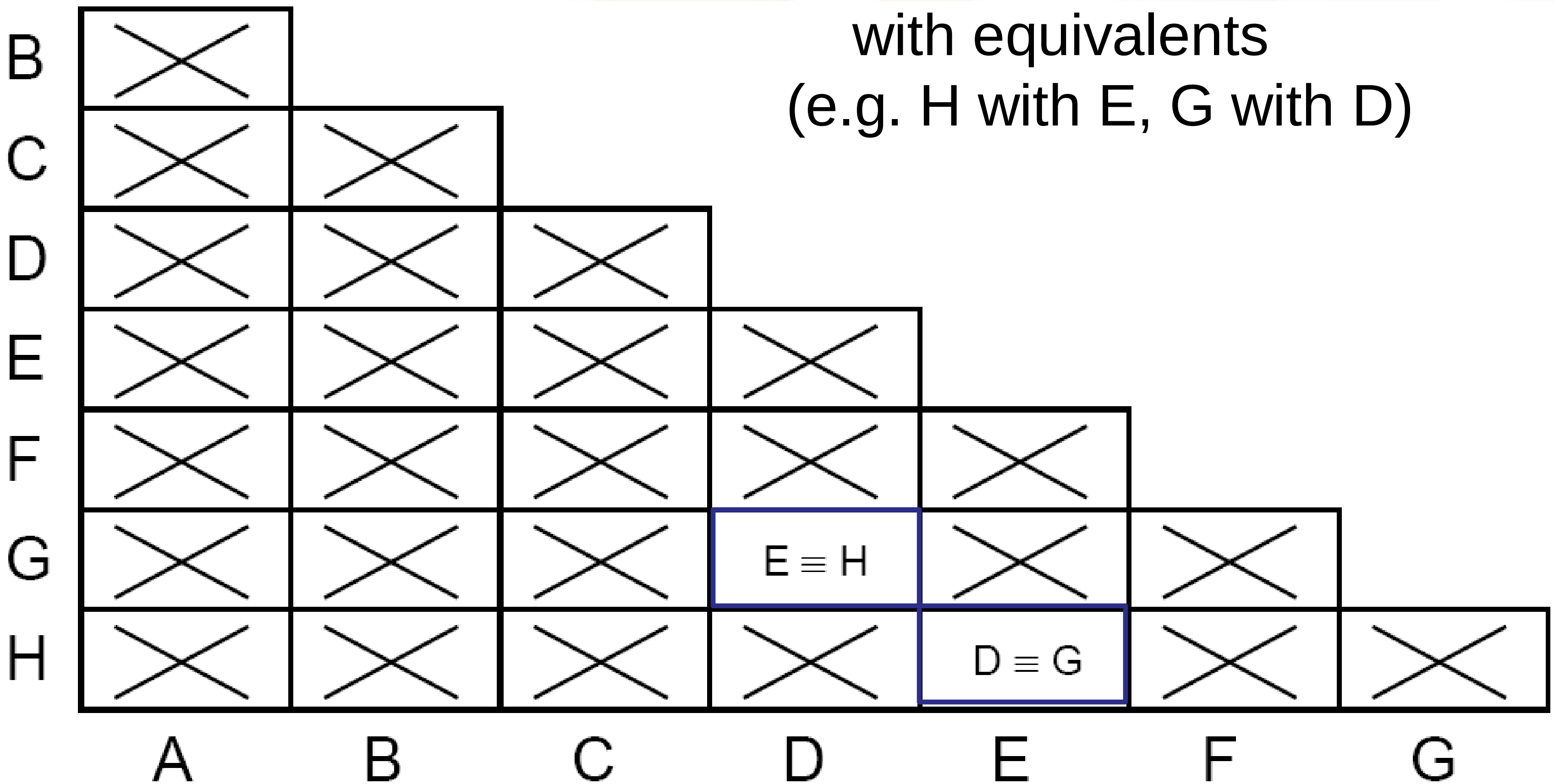
Implication Charts

5. Repeat until no more can be cancelled

	A	B	C	D	E	F	G
B	X						
C	~~C ≡ G~~ ~~B ≡ F~~	X					
D	~~B ≡ E~~	X	~~C ≡ G~~ ~~E ≡ F~~				
E	X	~~F ≡ H~~ ~~A ≡ G~~	X	X			
F	X	~~C ≡ F~~	X	X	~~C ≡ H~~ ~~A ≡ G~~		
G	~~D ≡ G~~ ~~B ≡ H~~	X	~~C ≡ D~~ ~~F ≡ H~~	E ≡ H	X	X	
H	X	~~E ≡ F~~ ~~A ≡ D~~	X	X	D ≡ G	~~C ≡ E~~ ~~A ≡ D~~	X

Implication Charts

6. What is left can be replaced with equivalents
(e.g. H with E, G with D)



Implication Charts

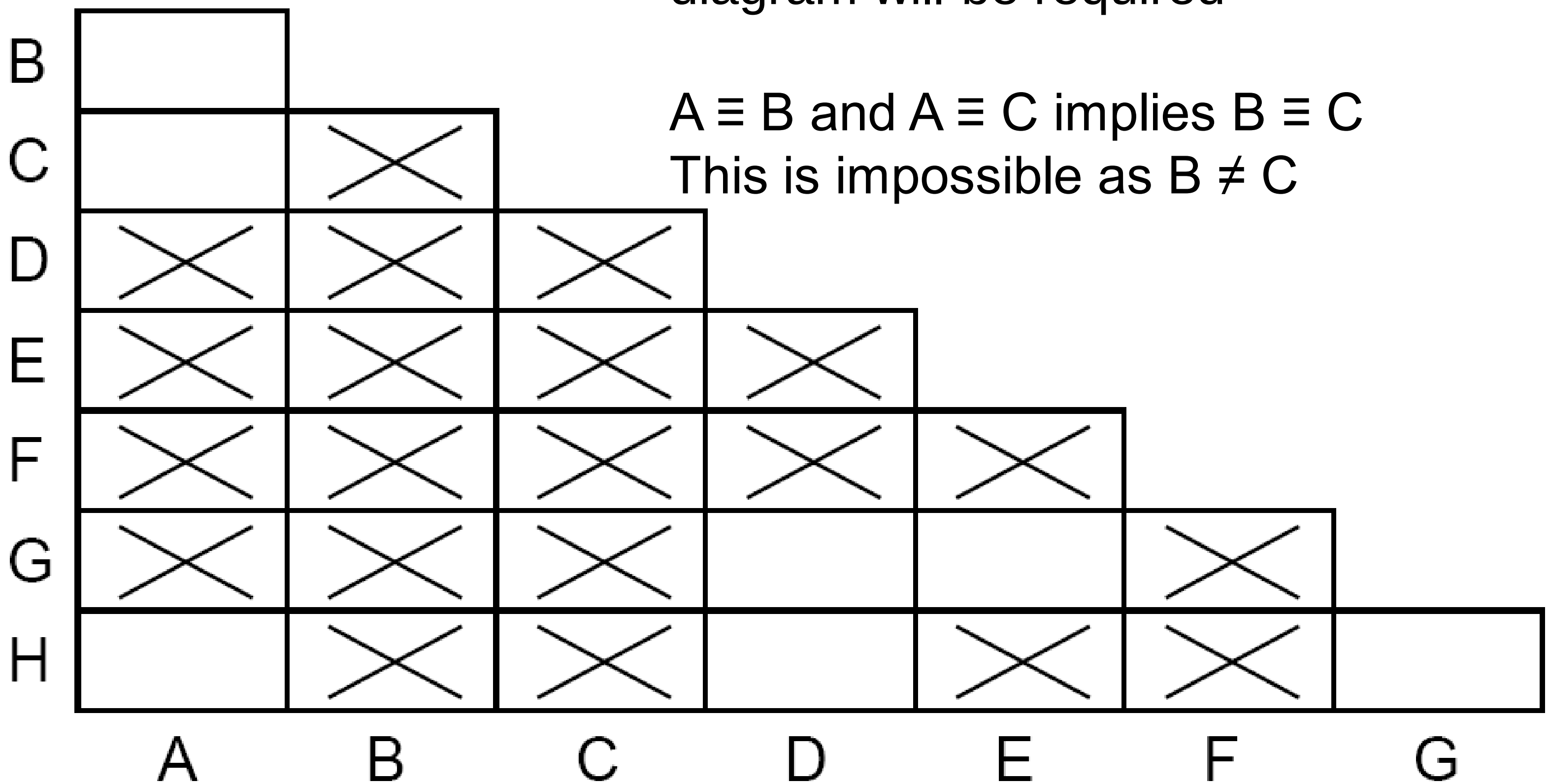
7. Draw reduced state table

Present state	Next state		Output Z	
	X=0	X=1	X=0	X=1
A	D	B	1	0
B	F	A	0	1
C	C	F	1	0
D	D	E	1	0
E	E	D	0	1
F	C	A	0	1

Merger Diagrams

Sometimes implication charts show many possible equivalences. If this occurs a Merger diagram will be required

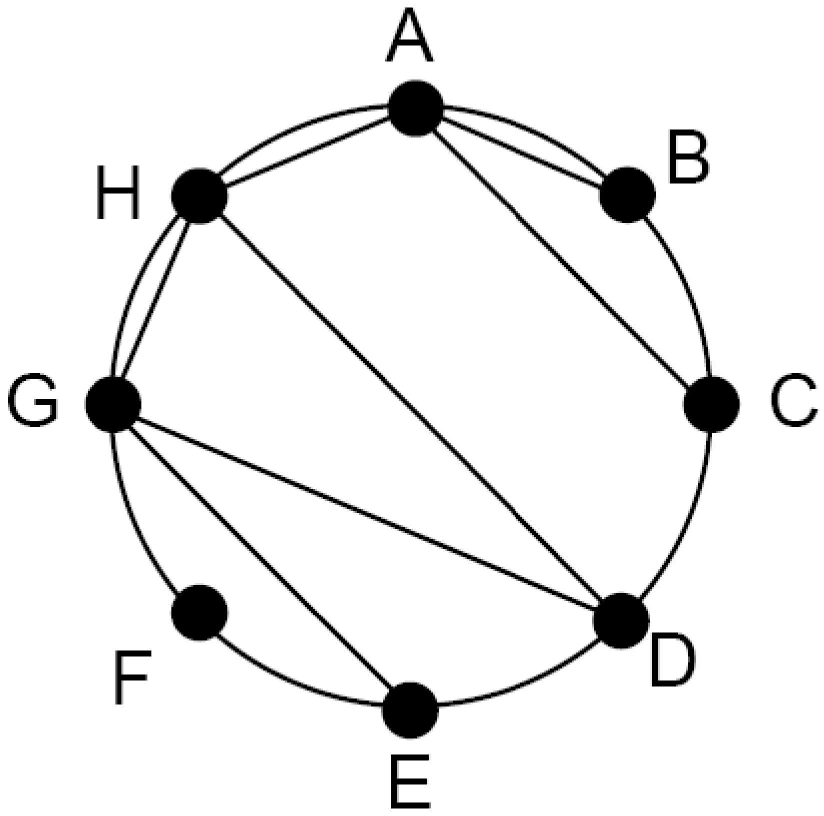
A ≡ B and A ≡ C implies B ≡ C
This is impossible as B ≠ C



A≡B A≡C A≡H D≡G D≡H E≡G G≡H

Merger Diagrams

- Lines are placed on the merger diagram with regards to all possible equivalences
- Polygons formed by these lines with all their sides displayed are to be found.
- The triangle GDH determines that G≡D≡H must be true



G≡H≡D

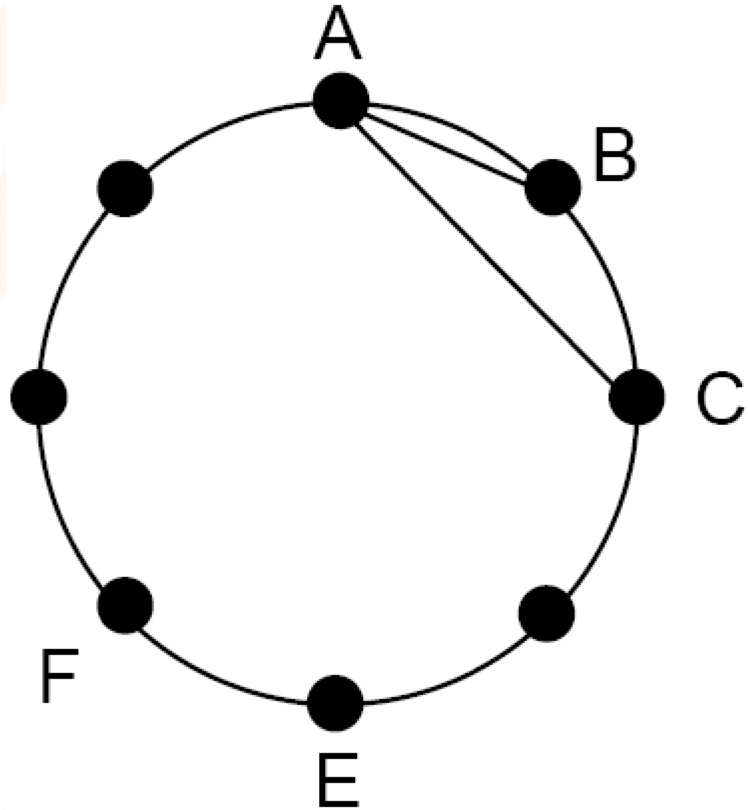
A ≡ B

A ≡ C

B ≠ C

Merger Diagrams

- We are left with an arbitrary decision between A≡B and A≡C



G≡H≡D
A ≡ B
A ≡ C
B ≠ C