

# CS2100 Comp Org Notes

AY23/24 Sem 1, [github.com/gerteck](https://github.com/gerteck)

## 12. Boolean Algebra

### Digital Circuits

- Two voltage levels, 1 for high, 0 for low.
- Digital circuits over analog circuits are more reliable, specified accuracy (determinable).
- Digital circuits abstracted using simple mathematical model: (**Boolean Algebra**)
- Design, Analysis and simplification of digital circuit: **Digital Logic Design**.
- Combinational:** no memory, output depends solely on the input. (gates, adders, multiplexers)
- Sequential:** with memory, output depends on both input and current state. (counters, registers, memories)

### Boolean Algebra

connectives in order of precedence:

- negation**  $A'$  equivalent to **NOT**
- conjunction**  $A \cdot B$  equivalent to **AND**
- disjunction**  $A + B$  equivalent to **OR**
- Note: always write the AND operator  $\cdot$ , do not omit, as it may be confused with a 2 bit value,  $AB$ .
- Truth Table:** Provides listing of every possible combination of inputs and corresponding outputs. We may prove using truth table by comparing columns.

### Duality

- Duality:** if the AND/OR operators and identity elements 0/1 interchanged in a boolean equation, it remains valid.
- e.g. the dual equation of  $a + (b \cdot c) = (a + b) \cdot (a + c)$  is  $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ , where if one is valid, then its dual is also valid.

## Laws & Theorems of Boolean Algebra

<b>Identity laws</b>	
$A + 0 = 0 + A = A$	$A \cdot 1 = 1 \cdot A = A$
<b>Inverse/complement laws</b>	
$A + A' = A' + A = 1$	$A \cdot A' = A' \cdot A = 0$
<b>Commutative laws</b>	
$A + B = B + A$	$A \cdot B = B \cdot A$
<b>Associative laws *</b>	
$A + (B + C) = (A + B) + C$	$A \cdot (B \cdot C) = (A \cdot B) \cdot C$
<b>Distributive laws</b>	
$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	$A + (B \cdot C) = (A + B) \cdot (A + C)$

<b>Idempotency</b>	
$X + X = X$	$X \cdot X = X$
<b>One element / Zero element</b>	
$X + 1 = 1 + X = 1$	$X \cdot 0 = 0 \cdot X = 0$
<b>Involution</b>	
$(X')' = X$	
<b>Absorption 1</b>	
$X + X \cdot Y = X$	$X \cdot (X + Y) = X$
<b>Absorption 2</b>	
$X + X' \cdot Y = X + Y$	$X \cdot (X' + Y) = X \cdot Y$
<b>DeMorgans'</b> (can be generalised to more than 2 variables)	
$(X + Y)' = X' \cdot Y'$	$(X \cdot Y)' = X' + Y'$
<b>Consensus</b>	
$X \cdot Y + X' \cdot Z + Y \cdot Z = X \cdot Y + X' \cdot Z$	$(X+Y) \cdot (X'+Z) \cdot (Y+Z) = (X+Y) \cdot (X'+Z)$

left/right equations are duals of each other

### Proving Theorems

- Theorems can be proved using truth table, or by algebraic manipulation using other theorems/laws.

\* Example: Prove absorption theorem  $X + X \cdot Y = X$

$$\begin{aligned} X + X \cdot Y &= X(1 + Y) \text{ (by identity law)} \\ &= X(1+Y) \text{ (by distributivity)} \\ &= X \cdot 1 \text{ (by one element law)} \\ &= X \text{ (by identity law)} \end{aligned}$$

\* By the principle of duality, we may also cite (without proof) that  $X \cdot (X+Y) = X$ .

## Boolean Functions, Complements

- Represented by  $F$ , e.g.  $F_1(x, y, z) = x \cdot y \cdot z'$ .
- To prove  $F_1 = F_2$ , we may use boolean algebra, or use truth tables.
- Complement Function is denoted as  $F'$ , obtained by interchanging 1 with 0 in function's output values.

## Standard Forms

- Literals:** A Boolean variable on its own or in its complemented form. (e.g.  $x, x'$ )
- Product Term:** A single literal or a logical product (AND,  $\cdot$ ) of several literals. (e.g.  $x, x \cdot y \cdot z'$ )
- Sum Term:** A single literal or a logical sum (OR +) of several literals. (e.g.  $A + B'$ )
- sum-of-products (SOP) expression:** A product term or a logical sum (OR +) of several product terms.
- product-of-sums (POS) expression:** A sum term or a logical product (AND) of several sum terms.
- Every boolean expr can be expressed in SOP/POS form.

### Minterms and Maxterms

- minterm** (of  $n$  variables): a product term that contains  $n$  literals from all the variables; denoted  $m_0$  to  $m[2^n - 1]$
- maxterm** (of  $n$  variables): a sum term that contains  $n$  literals from all the variables; denoted  $M_0$  to  $M[2^n - 1]$
- Each minterm is the complement ( $m_2' = M_2$ ) of its corresponding maxterm, vice versa.

x	y	Minterms		Maxterms	
		Term	Notation	Term	Notation
0	0	$x \cdot y'$	$m_0$	$x+y$	$M_0$
0	1	$x \cdot y$	$m_1$	$x+y'$	$M_1$
1	0	$x \cdot y'$	$m_2$	$x+y$	$M_2$
1	1	$x \cdot y$	$m_3$	$x+y'$	$M_3$

### Canonical Forms

- Canonical/normal form: a unique form of representation.
- Sum-of-minterms** = Canonical sum-of-products
- Product-of-maxterms** = Canonical product-of-sums

Given truth table:

x	y	z	F1	F2	F3
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	0	1	0	1	1
1	1	0	1	1	0
1	1	1	1	0	0

Sum of minterms:  $\sum m_1 + m_2 + m_3 = m_1 + m_2 + m_3$

Product of maxterms:  $\prod M_0 \cdot M_1 \cdot M_2 \cdot M_3 = (x+y) \cdot (x+y') \cdot (x+y) \cdot (x+y')$

Product of maxterms:  $\prod M_0 \cdot M_1 \cdot M_2 \cdot M_3 = (x+y) \cdot (x+y') \cdot (x+y) \cdot (x+y')$

Product of maxterms:  $\prod M_0 \cdot M_1 \cdot M_2 \cdot M_3 = (x+y) \cdot (x+y') \cdot (x+y) \cdot (x+y')$

- We can convert between sum-of-minterms and product-of-maxterms easily, by DeMorgan's.

# 13. Logic Gates & Simplification

## Logic Gates

- Fan-in: The number of inputs of a gate  $\geq 1, 2$ .
- Implement bool exp / function as logic circuit.

## Universal Gates

- universal gate:** can implement a complete set of logic.
- $\{AND, OR, NOT\}$  are a complete set of logic, sufficient for building any boolean function.
- $\{NAND\}$  and  $\{NOR\}$  themselves a complete set of logic. Implement NOT/AND/OR using only NAND or NOR gates.

## SOP and POS

- an SOP expression can be easily implemented using
  - 2-level AND-OR circuit or 2-level NAND circuit
- a POS expression can be easily implemented using
  - 2-level OR-AND circuit or 2-level NOR circuit

## Algebraic Simplification

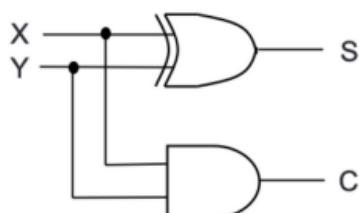
- Function Simplification:** Make use of algebraic (using theorems) or Karnaugh Maps (easier to use, limited to no more than 6 variables) or Quine-McCluskey.
- Algebraic Simplification:** aims to minimise
  - number of literals (prioritised over number of terms)
  - number of terms.

## Half Adder

- Half adder is a circuit that adds 2 single bits ( $X, Y$ ) to produce a result of 2 bits ( $C, S$ ).

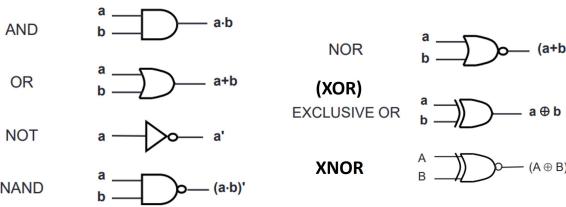
$$\circ C = X \cdot Y; \quad S = X \oplus Y$$

Inputs		Outputs	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

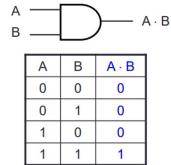


## Universal Gates

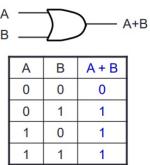
### Gate Symbols



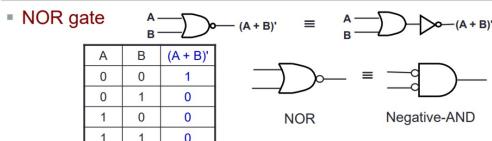
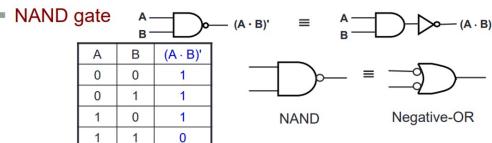
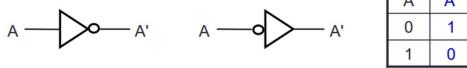
### AND Gate



### OR Gate



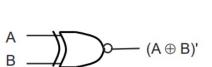
### (NOT gate)



### XOR gate

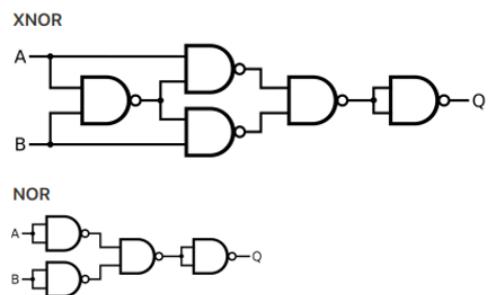
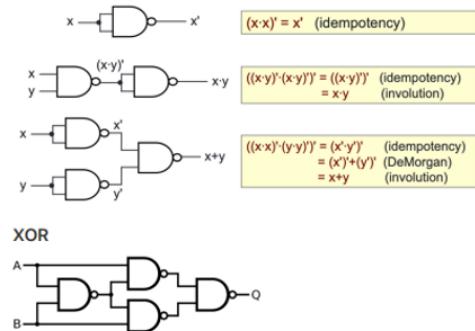


### XNOR gate



XNOR can be represented by  $\odot$   
(Example:  $A \odot B$ )

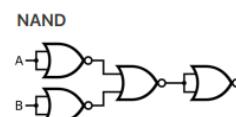
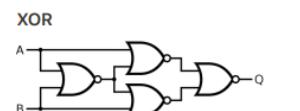
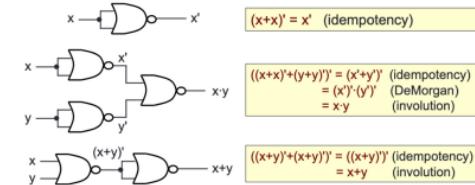
## NAND as Universal Gate (Complete Logic Set)



## NOR as Universal Gate (Complete Logic Set)

### NOR

- Proof: Implement NOT/AND/OR using only NOR gates.



## Gray Code

- Only a **single bit change** from one code value to the next.

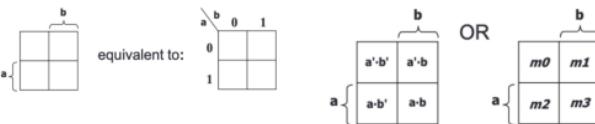
4 bit standard gray code:

Decimal	Binary	Gray Code	Decimal	Binary	Gray code
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

- not restricted to decimal digits:  $n$  bits can have up to  $2^n$  values.
- aka reflected binary code. To generate gray code, reflect.
- not unique - multiple possible Gray code sequences

## K Maps

- Simplify (SOP) expressions, with fewest possible product terms and literals.
- Based on **Unifying Theorem** ( $A + A' = 1$ ), **complement law**.
- Abstract form of Venn diagram, matrix of squares, each square represents a **minterm**.
- Two adjacent squares represent minterms that differ by exactly one literal.

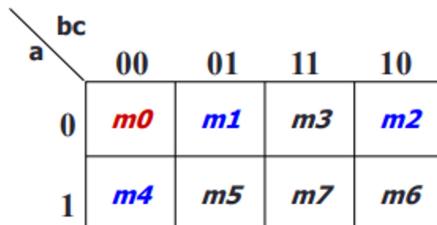


## K Map for a function:

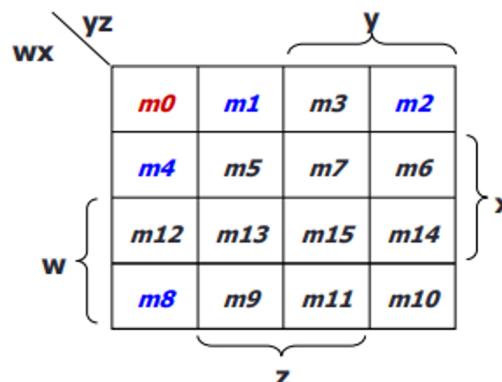
- The K-map for a function is filled by putting:
  - A '1' in the square that corresponds to a **minterm**
  - A '0' otherwise
- Each **valid grouping** of adjacent cells containing '1' corresponds to a simpler product term.
- Group must have width/length (size) in **powers of 2**.
- larger group** = fewer literals in result product term
- fewer groups** = fewer product terms in final SOP exp.
- Group maximum cells, and select fewest groups.

## K-Maps

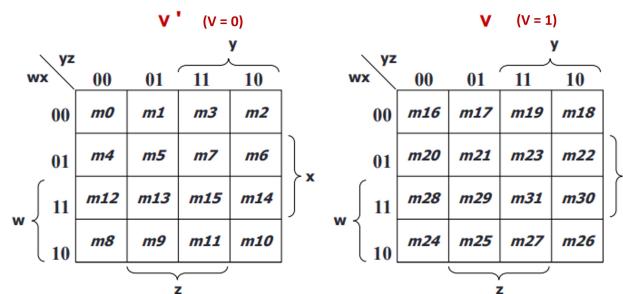
### 3-Variable



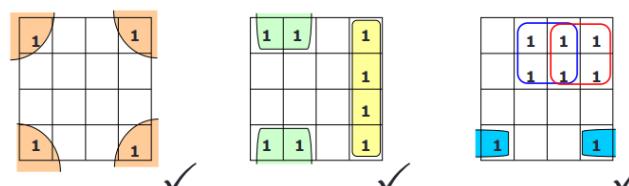
### 4-Variable



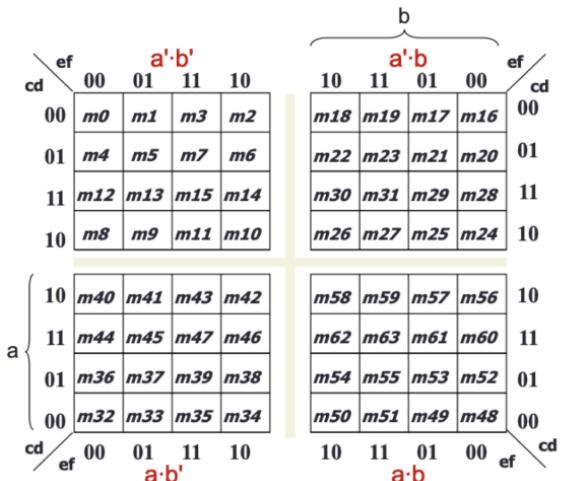
### 5-Variable



### Valid Groupings



## 6-Variable



### Using a K-map

- K-map of function easily filled in when function in sum-of-minterms form.
- If not in sum-of-minterms, convert into sum-of-products (SOP) form, expand SOP expr into sum-of-minterms, or fill directly based on SOP.

### (E)PIs

- implicant**: product term that could be used to cover minterms of the function.
- prime implicant**: a product term obtained by combining the maximum possible number of minterms from adjacent squares in the map.
- essential prime implicant**: a prime implicant that includes at least one minterm that is not covered by any other prime implicant

### K-maps to find POS

- shortcut: group maxterms (0s) of given function
- long way: 1. convert K-map of F to K-map of F' (by flipping 0/1s), 2. get SOP of F' POS=(SOP)'.

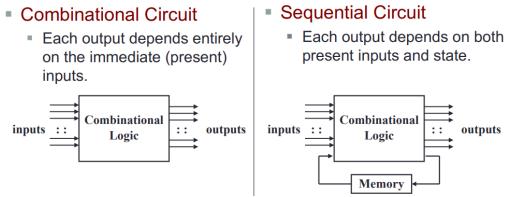
### Don't-Care Conditions

- denoted  $d$ , e.g.:  $F(A, B, C) = \sum m(3, 5, 6) + \sum d(0, 7)$

# 14. Combinational Circuits

## Combinational Circuits

- Two classes of logic circuits, combinational and sequential.



- Function analysis of combinational circuit (CC):** Label inputs and outputs, obtain functions of intermediate points and draw the truth table. Deduce functionality.
- CC design methods:** gate-level (with logic gates) and block-level (with functional blocks, e.g. IC chip).
- Goals: reduce cost, increase speed, design simplicity.

## Gate-Level (SSI: Small Scale Integration) Design

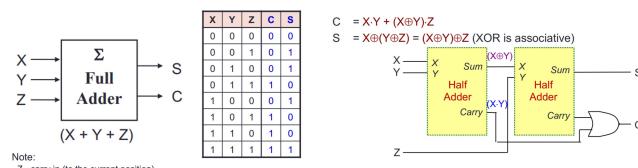
- Design procedure:
  - State problem, label input output of circuit.
  - Draw truth table, obtain simplified boolean function.
  - Draw logic diagram.

### Gate-Level (SSI) Design: Half Adder

$$\begin{aligned} \text{Example: } C &= X \cdot Y \\ S &= X' \cdot Y + X \cdot Y' = X \oplus Y \end{aligned}$$



### Gate-Level (SSI) Design: Full Adder



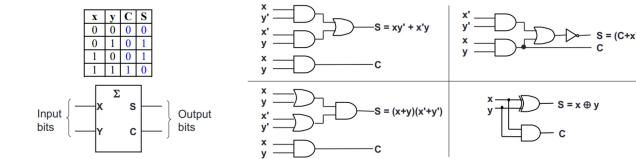
Using K-map, simplified SOP form:

$$\begin{aligned} C &= X \cdot Y + X \cdot Z + Y \cdot Z \\ S &= X' \cdot Y' \cdot Z + X \cdot Y' \cdot Z' + X \cdot Y \cdot Z + X \cdot Y \cdot Z' \end{aligned}$$

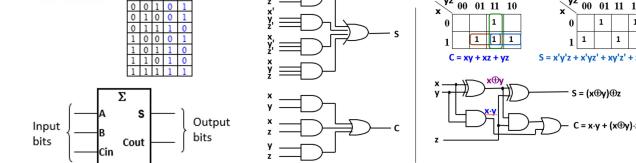
## Arithmetic Circuits Summary

- Block-Level Design** More complex circuits built using block-level method, rely on algorithm or formulae of circuit, decompose problem into solvable subproblems.

### Half adder

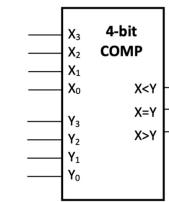
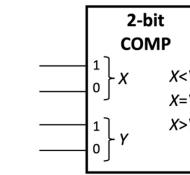


### Full adder

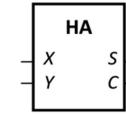


## Block Diagrams

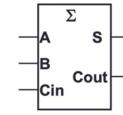
### Comparators



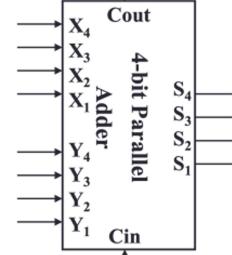
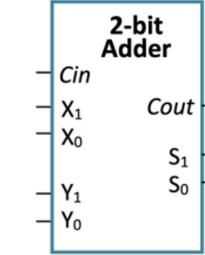
### Half Adder



### Full Adder

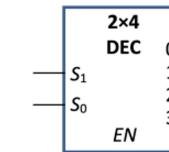


### Other Adders

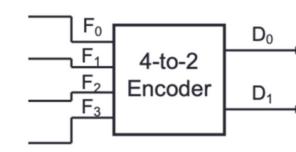


## MSI circuits Block Diagrams

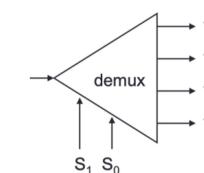
### Decoder



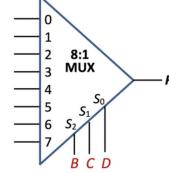
### Encoder



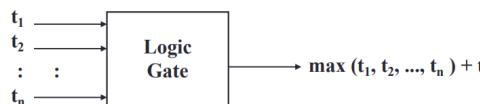
### Demultiplexer



### Multiplexer



- Given a **logic gate** with delay  $t$ . If inputs stable at times  $t_1, t_2, \dots, t_n$ , then earliest time in which output will be stable is:  $\max(t_1, t_2, \dots, t_n) + t$



- delay of a combinational circuit: repeat for all gates

- E.g. n-bit parallel adder will have delay of:

$$S_n = ((n-1) * 2 + 2)t$$

$$C_{n+1} = ((n-1) * 2 + 3)t$$

$$\text{max delay} = ((n-1) * 2 + 3)t$$

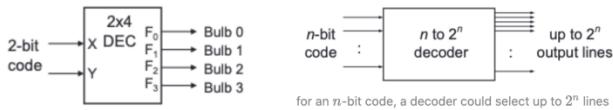
# 15. MSI Components

## Integrated Circuit

- **IC** aka chip or microchip is a set of electronic circuits on small flat piece of semiconductor.
- **Scale of Integration:** No. of components on standard size IC. (SSI: Small-scale Integration, MSI: Medium, LSI: Large, VLSI: Very large, ULSI: Ultra-large).

## Decoders

- convert binary information from  $n$  input lines, to up to  $2^n$  output lines
- selects only one output line, aka  $n$ -to- $m$ -line decoder,  $n : m$  or  $n \times m$  decoder where  $m \leq 2^n$ .



## Encoders

- given a set of input lines, of which exactly one is high and the rest are low, provide code that corresponds to the high input line.
- opposite of decoder.  $\leq 2^n$  input lines and  $n$  output lines
- implemented with OR gates



## Priority Encoders

- If multiple inputs are equal to 1, the highest **priority** takes precedence
- all inputs 0: invalid input
  - Example of a **4-to-2 priority encoder**:

Inputs				Outputs		
D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	f	g	v
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

## Enable

- **one-enable:** device is only activated when  $E = 1$
- **zero-enable:** device is only activated when  $E = 0$ , denoted or  $E'$  or  $\bar{E}$

E	X	Y	F <sub>0</sub>	F <sub>1</sub>	F <sub>2</sub>	F <sub>3</sub>
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	d	d	0	0	0	0

Decoder with 1-enable

E'	X	Y	F <sub>0</sub>	F <sub>1</sub>	F <sub>2</sub>	F <sub>3</sub>
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	1	0	0	0	1	0
0	1	1	0	0	0	1
1	d	d	0	0	0	0

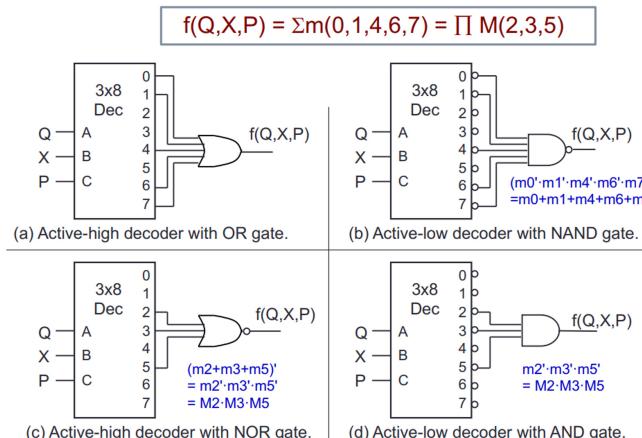
Decoder with 0-enable

## Zero-Enable/Negated Outputs

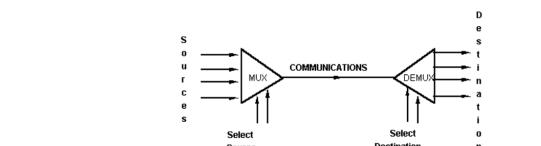
- **active-high outputs:** normal outputs (selected line is 1)
- **active-low outputs:** negated outputs (selected line is 0)

## Implementing Functions with Decoders

- any combinational circuit with inputs and outputs can be implemented with an  $n : 2^n$  decoder with  $m$  OR gates.
- **input:** bool function, in sum-of-minterms form
- **output:** decoder for minterms, OR gate to form sum.



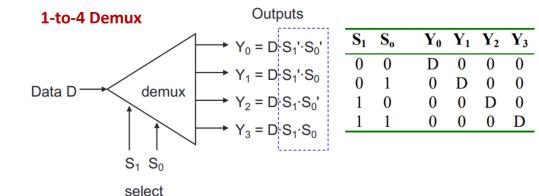
## Multiplexers & Demultiplexers



- Helps share **single comm line** among devices.
- One source, one dest at a time. (Circuit switching)

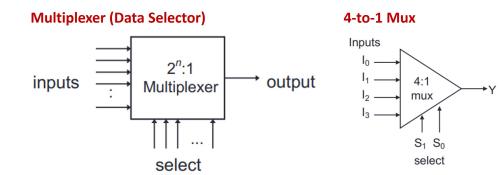
## Demultiplexer

- directs data from the input line to **one** selected output line
- **input:** an input line and set of selection lines
- **”output”:** directs data to one selected line
- **Identical to a decoder with enable**



## Multiplexer

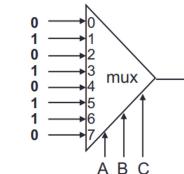
- steers one of  $2^n$  input lines to single output line, using selection lines.
- **input:** multiple input lines, multiple selection lines
- **output:** one output line = sum of the (product of data lines and selection lines)
- Larger multiplexers can be constructed from smaller ones.



## Implementing Functions with Multiplexers

- A  $2^n$ -to-1 multiplexer can implement bool function of  $n$  input variables:
- Express in sum-of-minterms form.
- Connect  $n$  variables to  $n$  selection lines, put ‘1’ on data line if minterm of function, ‘0’ otherwise

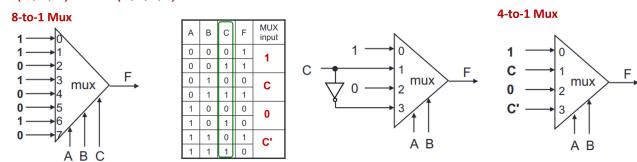
$$F(A,B,C) = A'B'C + A'B'C + A'B'C + A'BC' \\ = \Sigma m(1,3,5,6)$$



## Using smaller multiplexers for Functions

- We can use single smaller  $2^{(n-1)} - 1$  multiplexer to implement bool function of  $n$  (input) variables.
- Procedure:** Express function in sum-of-minterms form, reserve one variable (here take least significant one) for input lines of multiplexer, and use rest for selection lines. (C for input, A & B for selection)
- Draw truth table for function, group inputs by selection line values, determine multiplexer inputs by comparing input line (C) and function (F).

$$F(A,B,C) = \sum m(0,1,3,6) = A'B'C' + A'B'C + A'B'C + ABC'$$



## 16. Sequential Logic

- Sequential Circuit:** output depends on both present inputs and state
- 2 Types of sequential circuits:  
**synchronous:** outputs change only at a specific time  
**asynchronous:** outputs change at any time
- Classes of sequential circuits:  
**bistable:** 2 stable states (e.g. latches / flip-flops)  
**monostable:** 1 stable state; astable, no stable state

### Latches

- pulse-triggered (vs flipflops: edge-triggered)

### Flip-Flops

- synchronous bistable devices:** data on inputs is transferred to the flipflop's output only on the triggered edge of the clock pulse.
- output changes state at a specific point on the clock: change state at either rising or falling edge of the clock signal

### Memory Elements

- Memory Elements:** Device which can remember value indefinitely, or change value on command from its inputs.

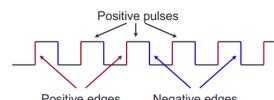
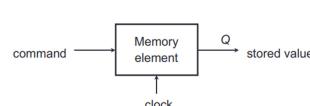
Characteristic table:

Command (at time $t$ )	$Q(t)$	$Q(t+1)$
Set	X	1
Reset	X	0
Memorise /	0	0
No Change	1	1

$Q(t)$  or  $Q$ : current state  
 $Q(t+1)$  or  $Q'$ : next state

### Memory Elements with Clock

- pulse-triggered:** ON = 1, OFF = 0
- edge-triggered:**
  - positive edge-trig:** ON: from 0 to 1, OFF: other t.
  - negative edge-trig:** ON: from 1 to 0, OFF: other t.



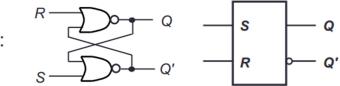
### Latches

A Latch is a **sequential circuit** that watches all inputs continuously and changes output at any time **independently of a clocking signal**.

#### S-R Latch

- Two **Inputs:**  $S$  (Set) and  $R$  (Reset).
- 2 complementary **outputs:**  $Q$  and  $Q'$
- When  $Q = \text{high}$ , latch is in SET state.
- When  $Q = \text{low}$ , latch is in RESET state.

▪ Active-high input S-R latch:



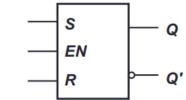
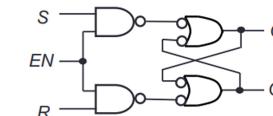
S	R	$Q(t+1)$	Description
0	0	$Q(t)$	No change. Latch remained in present state.
1	0	1	Latch SET.
0	1	0	Latch RESET.
1	1	0	Invalid condition.

S	R	$Q(t+1)$	Description
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	indeterminate	

#### Gated S-R Latch

- S-R Latch + enable input (EN) + 2 NAND gates**
- outputs change only when EN is high

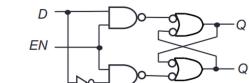
gated S-R latch.



#### Gated D Latch

- Gated D latch** make  $R$  input equal to  $S'$
- eliminates undesirable condition of invalid state in the S-R latch.

gated D latch



Characteristic table:

D	EN	$Q(t+1)$	Description
1	0	0	Reset
1	1	1	Set
0	X	$Q(t)$	No change

When  $EN=1$ ,  $Q(t+1) = D$

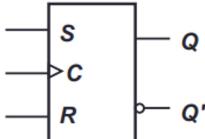
## Flip-flops

Flip-flops changes its outputs only at times determined by a clocking signal. AKA basic digital memory circuit.

### S-R flip flop

- > symbol at clock input. Negative edge-trigger:  $\circ >$ . Outputs change at triggering edge of clock pulse.

#### S-R Flip-flop



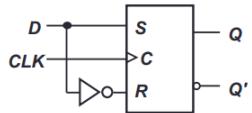
Positive edge-triggered flip-flop

S	R	CLK	$Q(t+1)$	Comments
0	0	X	$Q(t)$	No change
0	1	$\uparrow$	0	Reset
1	0	$\uparrow$	1	Set
1	1	$\uparrow$	?	Invalid

X = irrelevant ("don't care")

$\uparrow$  = clock transition LOW to HIGH

### D flip flop



A positive edge-triggered D flip-flop formed with an S-R flip-flop.

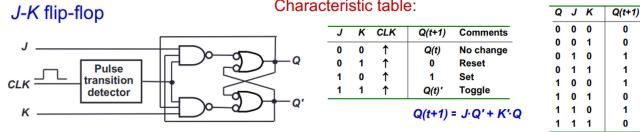
D	CLK	$Q(t+1)$	Comments
1	$\uparrow$	1	Set
0	$\uparrow$	0	Reset

$\uparrow$  = clock transition LOW to HIGH

### J-K flip flop

- No valid state, Includes **toggle** state:  $Q$  changes on each active clock edge.

#### J-K flip-flop



Characteristic table:

J	K	CLK	$Q(t+1)$	Comments
0	0	$\uparrow$	$Q(t)$	No change
0	1	$\uparrow$	0	Reset
1	0	$\uparrow$	1	Set
1	1	$\uparrow$	$Q(t)'$	Toggle

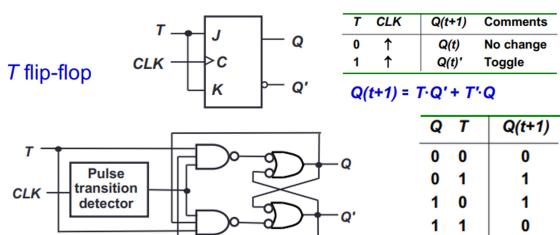
$$Q(t+1) = J \cdot Q(t)' + K \cdot Q(t)$$

### T flip flop

- T flip-flop:** Single input version of the J-K flip-flop, formed by tying both inputs together.

$$Q(t+1) = T \oplus Q$$

#### T flip-flop

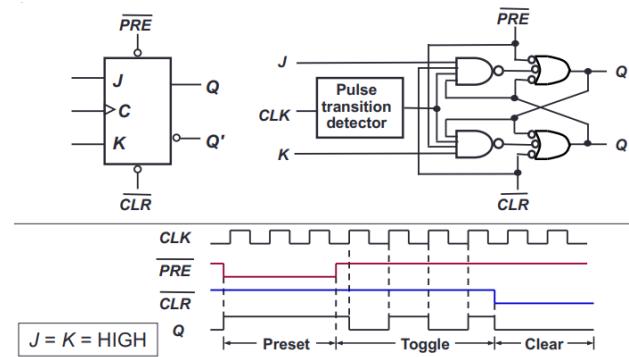


T	CLK	$Q(t+1)$	Comments
0	$\uparrow$	$Q(t)$	No change
1	$\uparrow$	$Q(t)'$	Toggle

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

## Asynchronous Inputs

- asynchronous** inputs affect the state of the flip-flop independent of the clock.
- e.g. J-K flip-flop with active-low PRESET and CLEAR asynchronous input



For **active-high**:

- PRE = 1 (HIGH): Q immediately set to HIGH
- CLR = 1 (HIGH): Q immediately cleared to LOW
- normal operation mode: both PRE and CLR are LOW

## Synchronous Sequential Circuits

- Building blocks: **logic gates & flip flops**.
- Flip-flops make up the memory, while gates form combinational sub-circuits.

## Flip Flop Characteristic Tables

Characteristic tables are used in analysis.

J	K	$Q(t+1)$	Comments
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q(t)'$	Toggle

S	R	$Q(t+1)$	Comments
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	?	Unpredictable

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

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

## Flip Flop Excitation Tables

Excitation tables are used in design.

Q	$Q^+$	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

JK Flip-flop

Q	$Q^+$	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

SR Flip-flop

Q	$Q^+$	D
0	0	0
0	1	1
1	0	0
1	1	1

D Flip-flop

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

T Flip-flop

## 17. Sequential Circuits Design

**Sequential Circuits Design:** (from state equations/table/diagram to logic circuit).

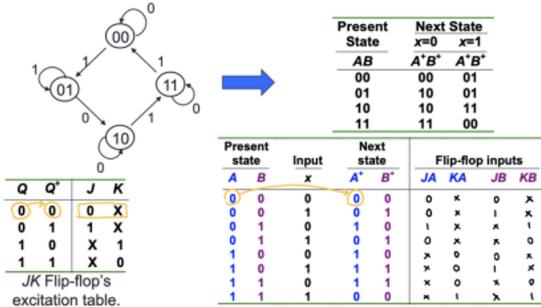
- For unused states, use **don't care X** for input.
- Self-correcting:** any unused state can transition to a used state after a finite number of cycles

- Convert state diagram to state table.
- fill in flip-flop inputs based on excitation table
- use K-maps to get simplified logic expressions for flip-flop inputs (e.g. JA, KA, JB, KB)
- draw circuit implementing

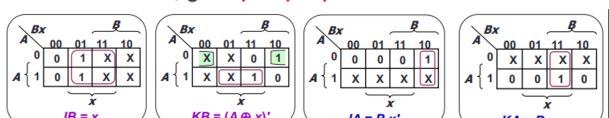
### Example

#### State diagram, design seq circuit with JK flip-flops

- Circuit state/excitation table, using JK flip-flops.



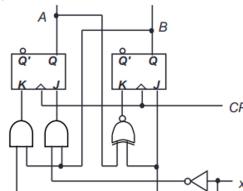
- From state table, get flip-flop input functions.



- Flip-flop input functions:

$$\begin{aligned} JA &= B \cdot x' \\ KA &= B \cdot x' \\ JB &= x' \\ KB &= (A \oplus x)' \end{aligned}$$

Logic Diagram:



### Unused States

- use don't-care for input/next state/flip-flop inputs/output.

Present state	Input	Next state	Flip-flop inputs	Output
A B C	x	A' B' C'	SA RA SB RB SC RC	y
0 0 1	0	0 0 1	0 X 0 X X 0	0
0 0 1	1	0 1 0	0 X 1 0 0 1	0
0 1 0	0	0 1 1	0 X X 0 1 0	0
0 1 0	1	1 0 0	1 0 0 1 0 X	0
0 1 1	0	0 0 1	0 X 0 1 X 0	0
0 1 1	1	1 0 0	1 0 0 1 0 1	0
1 0 0	0	1 0 1	X 0 0 X 1 0	0
1 0 0	1	1 0 0	X 0 0 X 0 X	1
1 0 1	0	0 0 1	0 1 0 X X 0	0
1 0 1	1	1 0 0	X 0 0 X X 0	1
1 1 0	0	0 0 0	X 0 0 X 0 1	1

Given these  
Unused state 000: ,10,111  
Derive these  
Are there other unused states?

Present state	Input	Next state	Flip-flop inputs
0 0 0	0	X X X X X X X X	
0 0 0	1	X X X X X X X X	

### Sequential Circuits Analysis

(from logic circuit to state equations/table/diagram).

- obtain flip-flop input functions from the circuit
- fill in the state table (using characteristic table)
- draw state diagram

$$\begin{aligned} JA &= B \\ KA &= B \cdot x' \\ JB &= x' \\ KB &= A' \cdot x + A \cdot x' = A \oplus x \end{aligned}$$

Fill the **state table** using the above functions, knowing the characteristics of the flip-flops used.

J K	Q(j+1)	Comments	Present state		Next state		Flip-flop inputs				
			A	B	x	A'	B'	JA	KA	JB	KB
0 0	0	No change	0	0	0	0	0	0	0	1	0
0 1	1	Reset	0	1	0	0	1	0	0	1	0
1 0	1	Set	1	0	1	1	0	1	1	1	0
1 1	Q(j+1) Toggling		0	1	1	0	1	0	0	0	1
0 0	0		0	0	0	0	1	0	0	1	0
0 1	0		0	1	0	1	1	0	0	1	1
1 0	0		1	0	0	1	0	1	0	0	1
1 1	0		1	1	1	1	1	0	0	0	0
0 0	1		0	0	1	0	1	1	1	1	1
0 1	1		0	1	1	1	0	0	0	0	0
1 0	1		1	0	0	1	1	0	0	1	1
1 1	1		1	1	1	1	1	1	1	1	1

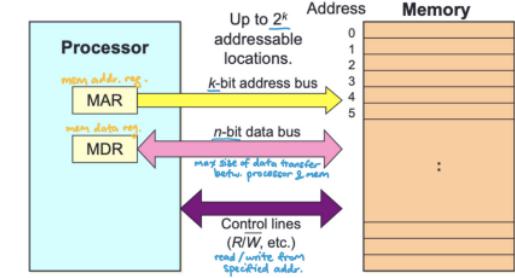
## 18. Memory

**Memory:** stores programs and data.

- 1 byte: 8 bits; 1 word: multiple of bytes, usually size of register
- memory unit stores binary info in words (groups of bits)
- data consists of n lines for n-bit words
  - data input lines:** provide info to write into memory
  - data output lines:** carries info to be read from memory
- address** consists of k lines
  - specifies which word (of the words available) to be selected for reading/writing
- control lines R/W** → specifies direction of transfer of data

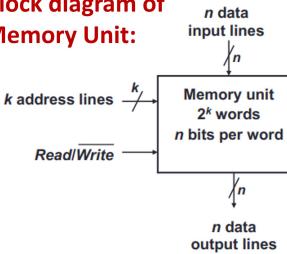
### Data Transfer

#### Data transfer



### Memory Unit, Read/Write Operations

#### Block diagram of Memory Unit:



### Write

- transfers address of the desired word to the address lines
- transfers the word (data bits) to be stored in memory to the data input lines
- activates the Write control (set R/W to 0)

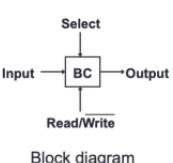
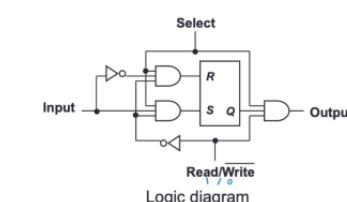
### Read

- transfers address of the desired word to the address lines
- activates the Read control (set R/W to 1)

### Memory Cell, Memory Array

#### • 2 types of RAM:

- static RAMs: use flip-flops as the memory cells
- dynamic RAMs: use capacitor charges to represent data
- Memory Arrays:** array of RAM chips, memory chips combined to form larger memory.



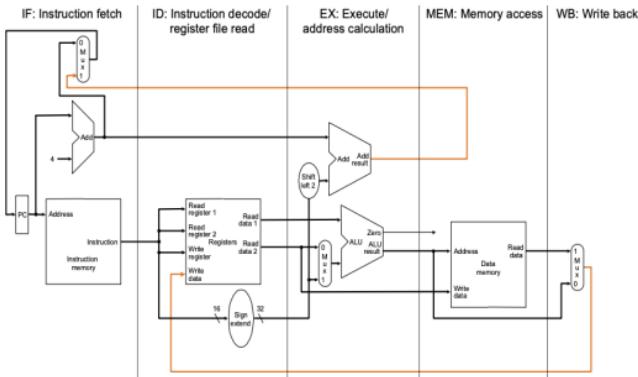
# 19. MIPS Pipelining

- Pipelining** improves throughput of entire workload, but does not help latency of a single task.
- Multiple tasks operating simultaneously using different resources.
- Pipeline rate limited by slowest pipeline stage, stall for dependencies.
- Pipeline Implementation of MIPS:**

- one cycle per pipeline stage.
- data required for each stage stored separately, as we need to carry data from one stage to next separately.

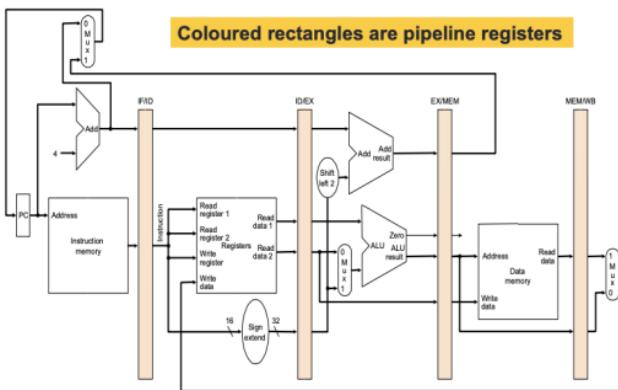
## MIPS Pipeline

- IF** - instruction fetch
- ID** - instruction decode and register read
- EX** - execute an operation or calculate an address
- MEM** - access an operand in data mem
- WB** - write result back to a register
- each stage is 1 clock cycle.
- exceptions: updating PC and writing back to register file



## Pipeline Registers

- Data used by same instruction in later pipeline stages are stored in pipeline registers.
- Each pipeline register is a collection of registers.
- Pipeline Registers:**
  - IF/ID** : register between IF and ID
  - ID/EX** : register between ID and EX
  - EX/MEM** : register between EX and MEM
  - MEM/WB** : register between MEM and WB
  - no register needed for the end of the WB stage



## Pipeline Control

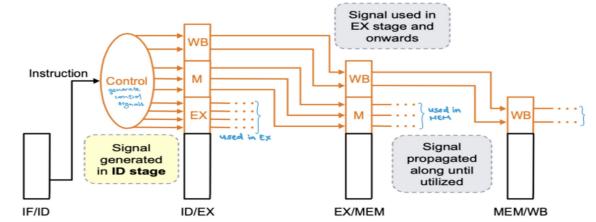
- Same control signals as single-cycle datapath.
- Each control signal belongs to a particular pipeline stage, avoid mixing up control from different pipelines.

## Control Grouping

- Group control signals according to pipeline stage.

	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop op1	ALUop op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

	EX Stage				MEM Stage			WB Stage	
	RegDst	ALUSrc	ALUop op1	ALUop op0	Mem Read	Mem Write	Branch	MemTo Reg	Reg Write
R-type	1	0	1	0	0	0	0	0	1
lw	0	1	0	0	1	0	0	1	1
sw	X	1	0	0	0	1	0	X	0
beq	X	0	0	1	0	0	1	X	0



## Pipeline Datapath

### 1. IF Stage

- at end of a cycle, **IF/ID** pipeline reg receives & stores:
- (PC + 4) & instruction from InstructionMemory[PC]

### 2. ID Stage

#### IF/ID register supplies:

- 2 Register numbers for reading registers
- 16-bit offset to be sign-extended to 32-bit.
- PC + 4

#### ID/EX register receives:

- Data values read from reg file
- 32-bit immediate value
- PC + 4

### 3. EX Stage

#### ID/EX register supplies:

- Data values read from reg file
- 32-bit immediate value
- PC + 4
- + write register number

#### EX/MEM register receives:

- (PC + 4) + (Immd × 4)
- ALU result
- isZero? signal
- RD2 from register file
- + write register number

### 4. MEM Stage

#### EX/MEM register supplies:

- (PC + 4) + (Immd × 4)
- ALU result
- mem read data
- + write register number

#### MEM/WB register receives:

- ALU result
- mem read data
- + write register number

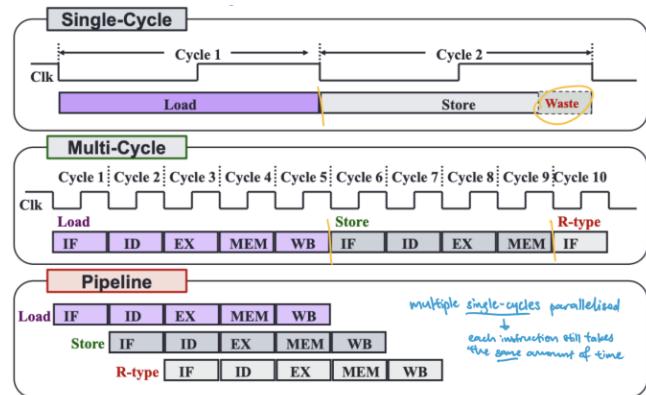
### 5. WB Stage

#### MEM/WB register supplies:

- ALU result
- mem read data
- write register number

**End of Cycle:** result is written back to register file if applicable.

# Pipeline Implementations



## Performance Comparison of Implementations

- $CT$  = cycle time
- $T_K$  = time for operation in stage
- $N$  = number of stages

### single-cycle processor

- cycle time:  $CT_{seq} = \max(\sum_{k=1}^{N-1} T_K)$
- execution time for  $I$  instructions  $I \times CT_{seq}$

### multi-cycle processor

- cycle time:  $CT_{multi} = \max(T_K)$
- execution time for  $I \times \text{Average CPI} \times CT_{multi}$
- (average CPI needed as each instr. takes diff no. of cycles.)

### pipelined processor

- cycle time:  $CT_{pipeline} = \max(T_K) + T_d$
- $T_d$  = overhead for pipelining (e.g. pipeline register)
- cycles needed for  $I$  instructions =  $(I + N - 1)$
- execution time for  $I$  instructions  $(I + N - 1) \times CT_{pipeline}$

## Pipelined Ideal Speedup

### assumptions:

- Every stage takes same amount of time:  $\sum_{k=1}^{N-1} = N \times T_1$
- No pipeline overhead  $T_d = 0$
- $I \gg N$  (num. of instructions much larger num. of stages)

$$\text{Speedup}_{\text{pipeline}} = \frac{\text{Time}_{\text{seq}}}{\text{Time}_{\text{pipeline}}} = N$$

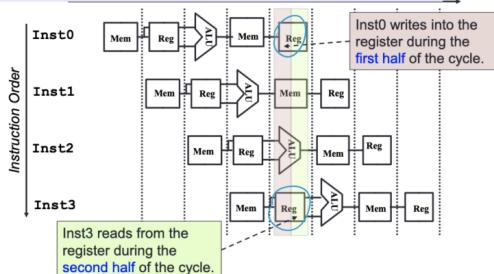
## 20. Pipeling Hazards

- **Pipeline hazards** prevent next instruction from immediately following previous instruction.
- **Structural Hazards:** simultaneous use of hardware resource.
- **Data Hazards:** data dependencies between instructions.
- **Control Hazards:** change in program flow.

### Structural Hazards

- solves read/write conflict in MEM module.
- split cycle into half.
- first half: write into register
- second half: read from register.

Recall that registers are very fast memory.  
Solution: Split cycle into half; first half for writing into a register; second half for reading from a register.

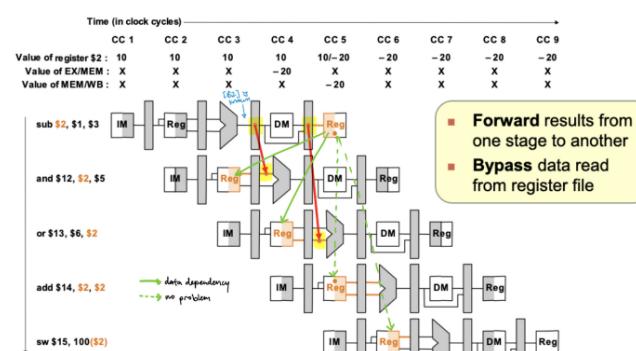


### Data Hazards

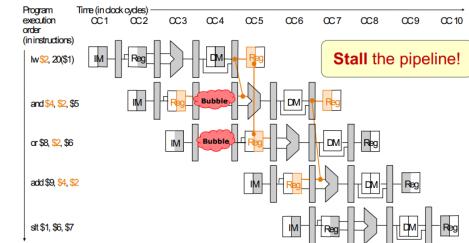
- instruction dependency: read-after-write (RAW)
- WAR, WAW data dep. do not cause pipeline hazards.

### Forwarding

- forward data right after EX stage completed, bypass (replace) data read from register file.
- usually no delay incurred



- for  $lw$ , data is only ready after MEM stage.
- No choice, may incur extra 1 cycle delay.

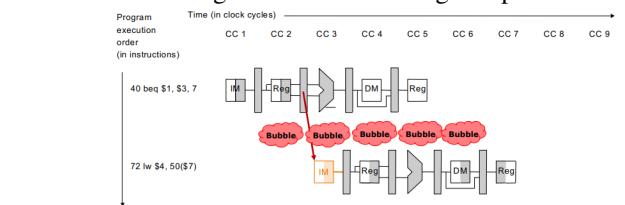


### Control Hazards

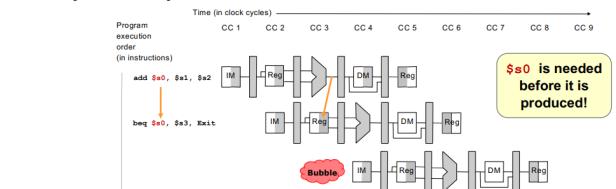
- branch decision made in MEM stage: **stall pipeline**
- 3 cycles delay (next IM starts after curr. MEM)

### Early Branch Resolution

- Make decision in ID stage instead of MEM.
- Move branch target address calc & reg comparison.



- 1 cycle delay



- 2 cycle delay if there is data dependency as well.
- 2 cycle delay if  $lw$  occurs before branch (wait stage 4)

### Branch Prediction (assume branch not taken)

- **if branch taken (prediction wrong):**  
stall 1 or 3 cycles depending on early branching.

### Delayed Branch

- if branch outcome takes  $X$  no. of cycles to be known, **move non-control dependent** instr into the  $X$  slots following a branch.
- aka branch delay slot ('no-op' nop slots), else fill w. nop.
- instructions executed regardless of branch outcome

## 21. Cache