CS2100 Summary Sheet (Part II)

Laws of Boolean Algebra

Identity laws	
A + 0 = 0 + A = A	A · 1 = 1 · A = A
Inverse/complement laws	
A + A' = A' + A = 1	A · A' = A' · A = 0
Commutative laws	
A + B = B + A	A · B = B · A
Associative laws *	
A + (B + C) = (A + B) + C	A · (B · C) = (A · B) · C
Distributive laws	
A · (B + C) = (A · B) + (A · C)	A + (B · C) = (A + B) · (A + C)

Theorems

Idempotency	
X + X = X	X · X = X
One element / Zero element	
X + 1 = 1 + X = 1	X · 0 = 0 · X = 0
Involution	
(X')' = X	
Absorption 1	
X + X·Y = X	X·(X + Y) = X
Absorption 2	
X + X'·Y = X + Y	X·(X' + Y) = X·Y
DeMorgans' (can be generalised to more than 2 variables)	
(X + Y)' = X' · Y'	(X · Y)' = X' + Y'
Consensus	
X·Y + X'·Z + Y·Z = X·Y + X'·Z	(X+Y)·(X'+Z)·(Y+Z) = (X+Y)·(X'+Z)

Duality

If the AND/OR operators and identity elements 0/1 in a Boolean equation are interchanged, it remains valid.

`a+(b.c) = (a+b).(a+c)` == `a.(b+c) = (a.b)+(a.c)`

`x+1 = 1` == `x.0 = 0`

Literals

- A Boolean variable on its own or in its complemented form
- Examples: **`x, x', y, y'`**

Product Term

- A single literal or a logical product (AND) of several literals
- Examples: **`x, x.y.z', A'.B, A.B, d.g'.v.w`**

Sum Term

- A single literal or a logical sum (OR) of several literals
- Examples: **`x, x+y+z', A'+B, A+B, c+d+h'+j`**

Standard Forms

Sum-of-Products (SOP) Expression

- A product term or a logical sum (OR) of several product terms
- Examples: **`x, x+(y.z'), (x.y')+(x'.y.z), (A.B)+(A'.B), (A)+(B'.C)+(A.C')+(C.D)`**

Product-of-Sums (POS) Expression

- A sum term or a logical product (OR) of several sum terms
- Examples: **`x, x.(y+z'), (x+y')+(x'+y+z), (A+B).(A'+B), (A+B+C).D'.(B'+D+E')`**

	Expression	SOP?	POS?
(1)	X'·Y + X·Y' + X·Y·Z	✓	×
(2)	(X+Y')·(X'+Y)·(X'+Z')	×	✓
(3)	X' + Y + Z (sum term)	✓	✓
(4)	X·(W' + Y·Z)	×	×
(5)	X·Y·Z' (product term)	✓	✓
(6)	W·X'·Y + V·(X·Z + W')	×	×

*Every Boolean expression can be expressed in SOP or POS form.

Minterms and Maxterms

A **minterm** of n variables is a product term that contains n literals from all the variables.

- Example: On 2 variables x and y, the minterms are:
  x’.y’, x’.y, x.y’ and x.y

A **maxterm** of n variables is a sum term that contains n literals from all the variables.

- Example: On 2 variables x and y, the minterms are:
  x’+y’, x’+y, x+y’ and x+y

In general, with n variables we have up to 2^n minterms and 2^n maxterms.

x	y	Minterms		Maxterms	
		Term	Notation	Term	Notation
0	0	x'·y'	m0	x+y	M0
0	1	x'·y	m1	x+y'	M1
1	0	x·y'	m2	x'+y	M2
1	1	x·y	m3	x'+y'	M3

Important Fact: Each minterm is the complement of its corresponding maxterm. Likewise, each maxterm is the complement of its corresponding minterm.

```
m2 = x.y’
m2’ = (x.y’)’ = x’+y = M2
```

Canonical Forms

Canonical/Normal Form: A unique form of representation

- Sum-of-minterms = Canonical sum-of-products
- Product-of-maxterms = Canonical product-of-sums

Precedence of Operators

NOT > AND > OR

Sum-of-Minterms

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	0	1
1	0	0	0	1	1
1	0	1	0	1	1
1	1	0	1	1	0
1	1	1	0	1	0

F1 = x·y·z' = m6

F2 = x'·y'·z + x·y'·z' + x·y'·z + x·y·z' + x·y·z
= m1 + m4 + m5 + m6 + m7 = Σm(1,4,5,6,7) or Σm(1,4 – 7)

F3 = x'·y'·z + x'·y·z + x·y'·z' + x·y'·z
= m1 + m3 + m4 + m5 = Σm(1,3,4,5) or Σm(1,3 – 5)

Obtain **sum-of-minterms** expression by gathering the minterms of the function (where output is 1).

Product-of-Maxterms

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	0	1
1	0	0	0	1	1
1	0	1	0	1	1
1	1	0	1	1	0
1	1	1	0	1	0

F2 = (x+y+z) · (x+y'+z) · (x+y'+z')
= M0 · M2 · M3 = ΠM(0,2,3)

F3 = (x+y+z) · (x+y'+z) · (x'+y'+z) · (x'+y'+z')
= M0 · M2 · M6 · M7 = ΠM(0,2,6,7)

Obtain **product-of-maxterms** expression by gathering the maxterms of the function (where output is 0).

Conversion of Standard Forms

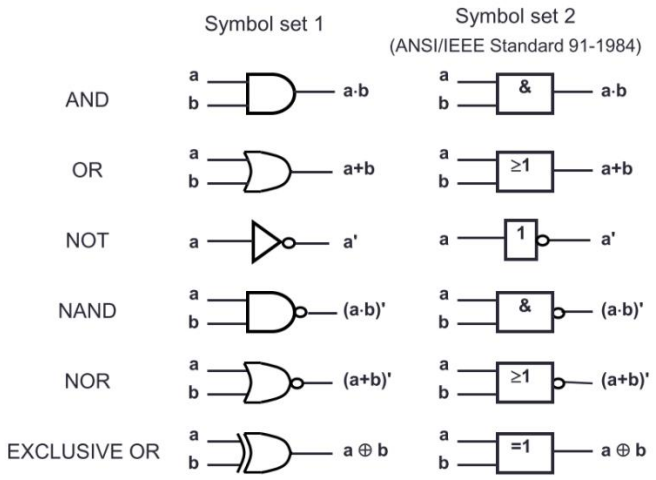
We can convert between SOM and POM easily

Example: F2 = $\sum$m(1,4,5,6,7) = ΠM(0,2,3)

```
F2’ = m0 + m2 + m3
F2 = (m0 + m2 + m3)’ = m0’ . m2’ . m3’ = M0 . M2 . M3
```

14. Logic Circuits

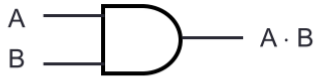
Gate Symbols



Inverter (NOT gate)

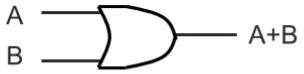
A	A'
0	1
1	0

AND gate



A	B	A · B
0	0	0
0	1	0
1	0	0
1	1	1

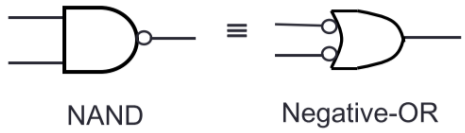
OR gate



A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

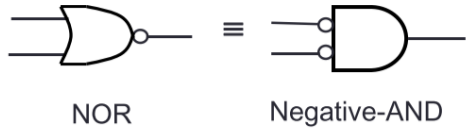
NAND gate

A	B	(A · B)'
0	0	1
0	1	1
1	0	1
1	1	0

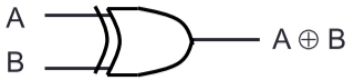


NOR gate

A	B	(A + B)'
0	0	1
0	1	0
1	0	0
1	1	0

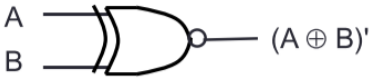


XOR gate



A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

XNOR gate



XNOR can be represented by ⊙
(Example: A ⊙ B)

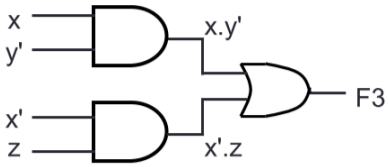
A	B	(A ⊕ B)'
0	0	1
0	1	0
1	0	0
1	1	1

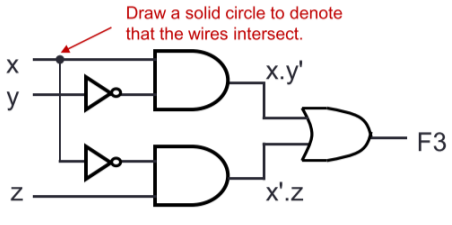
Logic Circuits

Fan-In: The number of inputs of a gate. Gates may have fun-in more than 2.

Example: a 3-input AND gate

Example: F3 = x·y' + x'·z





Universal Gates

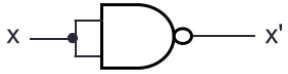
AND/OR/NOT gates are sufficient for building any Boolean function.
We call the set {AND, OR, NOT} a complete set of logic. Also, {NAND}, {NOR}, {AND,NOT}, {OR,NOT}.

However, other gates are also used:

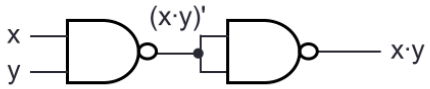
- Usefulness (XOR gate for parity bit generation)
- Economical
- Self-sufficient (NAND/NOR)

Universal Gates: NAND

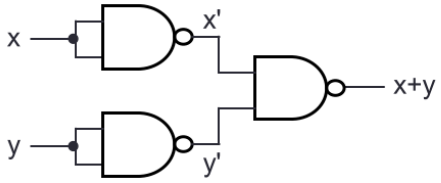
{NAND} is a complete set of logic.
Proof: Implement NOT/AND/OR using only NAND gates.



(x·x)' = x' (idempotency)



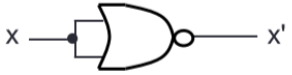
((x·y)'·(x·y)')' = ((x·y)')' (idempotency)
= x·y (involution)



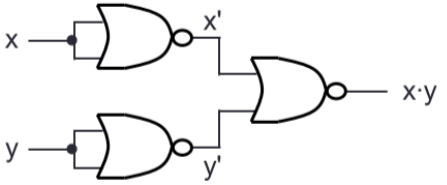
((x·x)'·(y·y)')' = (x'·y')' (idempotency)
= (x')'+(y')' (DeMorgan)
= x+y (involution)

Universal Gates: NOR

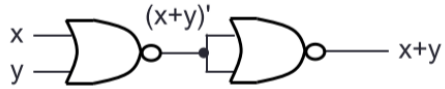
{NOR} is a complete set of logic.
Proof: Implement NOT/AND/OR using only NAND gates.



(x+x)' = x' (idempotency)



((x+x)'+(y+y)')' = (x'+y')' (idempotency)
= (x')'·(y')' (DeMorgan)
= x·y (involution)



((x+y)'+(x+y)')' = ((x+y)')' (idempotency)
= x+y (involution)

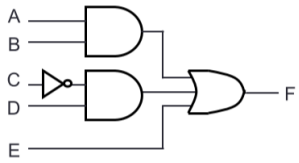
SOP and NAND Circuits

An SOP expression can be easily implemented using

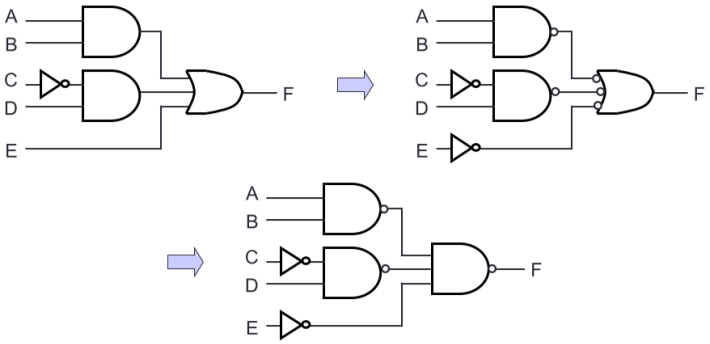
- 2-level AND-OR circuit
- 2-level NAND circuit

Example: **F = A.B + C'.D + E**

Using 2-level AND-OR circuit



Using 2-level NAND circuit



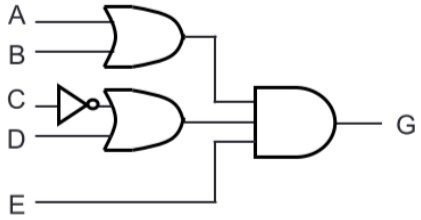
POS and NOR Circuits

A POS expression can be easily implemented using

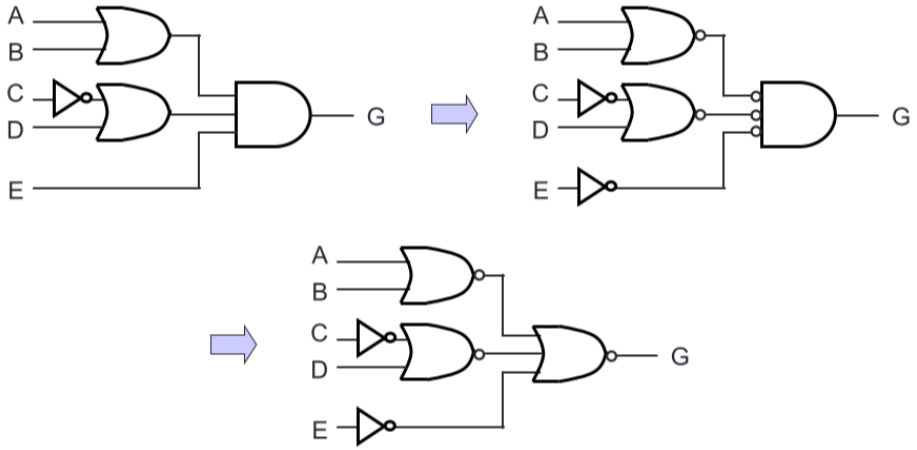
- 2-level OR-AND circuit
- 2-level NOR circuit

Example: **G = (A+B).(C'+D).E**

Using 2-level OR-AND circuit



Using 2-level NOR circuit



15. Simplification

Techniques:

- **Algebraic**
  - Using theorems
  - Open-ended; requires skills

Gray Code

- **Unweighted** (not an arithmetic code)
- Only a **single bit change** from one code value to the next
- Not restricted to decimal digits: n bits => 2^n values
- Good for error detection
- Named after Frank Gray; also called reflected binary code
- Example: 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

There are many gray code sequences. Example: 3 bits,

			These are NOT Gray codes (why?)		
000	000	110	000	010	010
001	010	111	001	110	011
011	110	101	010	101	111
010	111	100	011	001	110
110	011	000	100	100	111
111	001	001	101	111	101
101	101	011	110	000	001
100	100	010	111	011	000
			✗	✗	✗

This is the standard Gray Code.

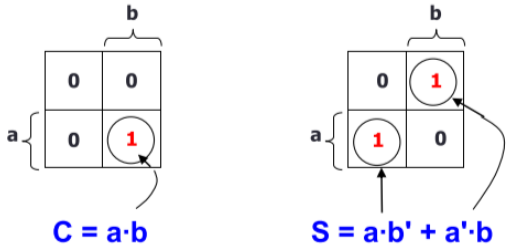
K-Maps

Systematic method to obtain **simplified sum-of-products (SOP) expressions**

- Objective: Fewest possible product terms and literals
- Advantage: Easy to use
- Disadvantage: Limited to 5 or 6 variables
- Each square represents a minterm
- Two adjacent squares represent minterms that differ by exactly one literal

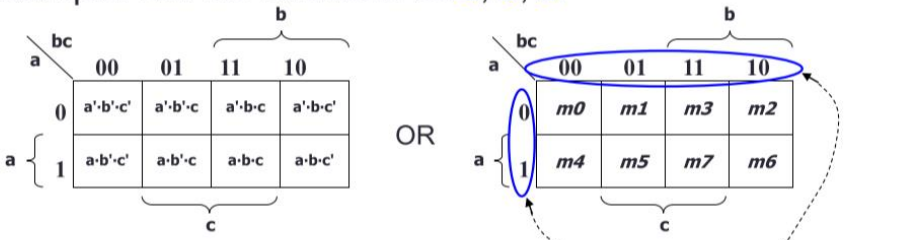
2-Variable K-Maps

Example: Half Adder



3-Variable K-Maps

- Example: Let the variables be a, b, c.

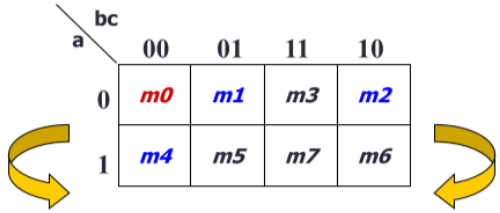


Above arrangement ensures that minterms of adjacent cells differ by only *ONE literal*. (Other arrangements which satisfy this criterion may also be used.)

Note *Gray code* sequence
A Gray code sequence is one where the value differs from its previous by 1 bit.

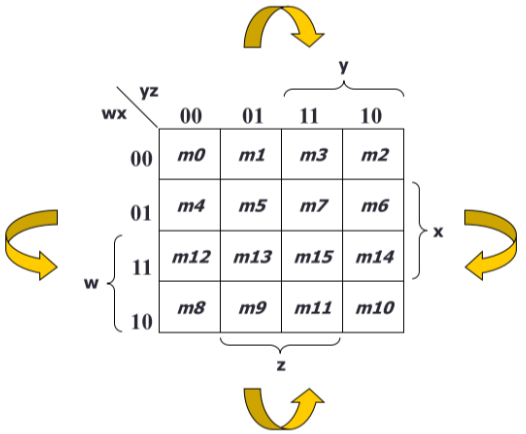
There is wrap-around in the K-map:

- a'·b'·c' (*m0*) is adjacent to a'·b·c' (*m2*)
- a·b'·c' (*m4*) is adjacent to a·b·c' (*m6*)



- Each cell in an n-variable K-map has n adjacent neighbours.
- m0 has 3 adjacent neighbours, m1, m2, and m4.

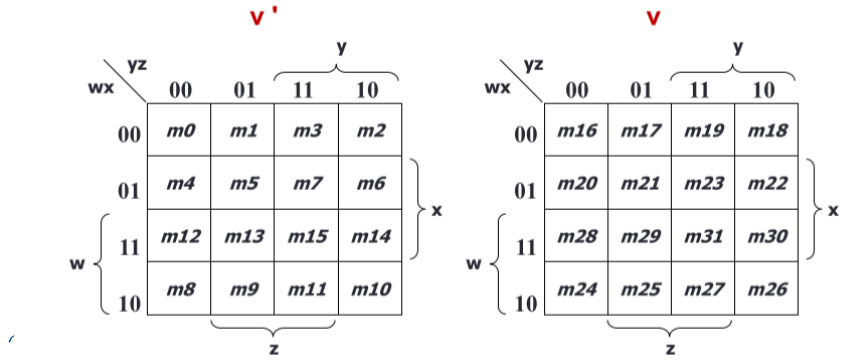
4-Variable K-Maps



- There are 2 wrap-arounds
- Every cell has 4 neighbours
  - The cell corresponding to minterm m0 has neighbours m1, m2, m4, and m8.

5-Variable K-Maps

Organised as two 4-variable K-maps. One for v' and the other for v.



How to use K-maps

- Based on the **Unifying Theorem** (complement law): **`A + A' = 1`**
- In a K-Map, each cell containing a '1' corresponds to a minterm of a given function F where the output is 1.
- Each valid grouping of adjacent cells containg '1' then corresponds to a simpler product term of F.
  - A group must have size in powers of two: 1, 2, 4, 8, …
  - Grouping 2^n cells eliminates n variables.
- Group as many cells as possible
  - The larger the group, the fewer the number of literals in the resulting product term
- Select as few groups as possible to cover all the cells (minterms) of the function
  - The fewer the groups, the fewer is the number of product terms in the simplified SOP expression

Example:

Here, there are 2 groups of minterms, A and B:

A = w'.x.y'.z' + w'.x.y'.z = w'.x.y'.(z' + z) = **w'.x.y'**

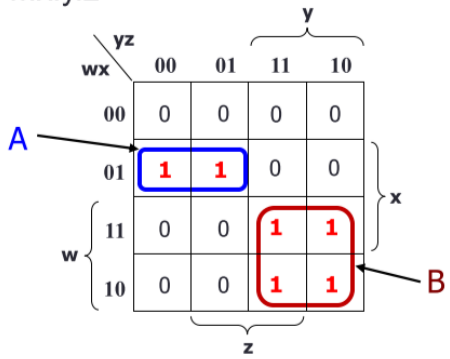
B = w.x'.y.z' + w.x'.y.z + w.x.y.z' + w.x.y.z

= w.x'.y.(z' + z) + w.x.y.(z' + z)

= w.x'.y + w.x.y

= w.(x'+x).y

= **w.y**



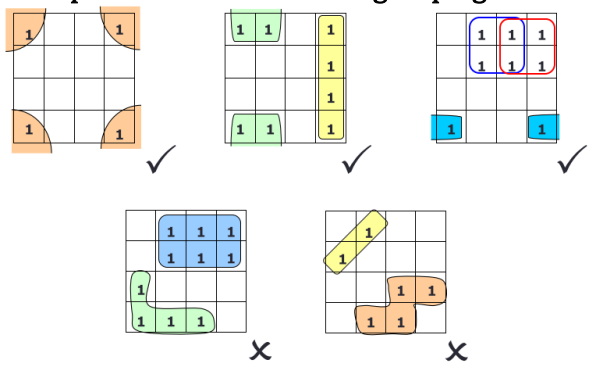
Each product term that corresponds to a group represents the sum of minterms in that group. Boolean expression is therefore the sum of product terms (SOP) that represent all groups of the minterms:

`F(w,x,y,z) = group A + group B = w'.x.y' + w.y`

The larger the group, the fewer is the number of literals in the associated product term. Ect: For a 4-var kmap with variables w,x,y,z

- 1 cell = 4 literals. Ect: w.x.y.z
- 2 cell = 3 literals. Ect: w.x.y
- …
- 16 cells = no literals. Ect. 1

Examples of valid and invalid groupings



Converting to Minterms Form

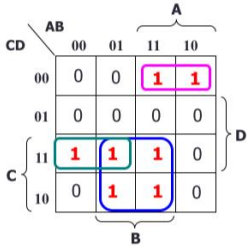
The K-map of a function can be easily filled in when the function is given in sum-of-minterms form. If it is not in SOM form, convert it into SOP form, then expand the SOP expression into SOM expression or fill in the KMap directly based on the SOP expression.

- Example:

F(A,B,C,D) = A.(C+D)'.(B'+D') + C.(B+C'+A'.D)
= A.(C'.D').(B'+D') + B.C + C.C' + A'.C.D
= A.B'.C'.D' + A.C'.D' + B.C + A'.C.D

Expanding it to sum of minterms (unnecessary):

A.B'.C'.D' + A.C'.D' + B.C + A'.C.D
= A.B'.C'.D' + A.C'.D'.(B+B') + B.C + A'.C.D
= A.B'.C'.D' + A.B.C'.D' + A.B'.C'.D' + B.C.(A+A') + A'.C.D
= A.B'.C'.D' + A.B.C'.D' + A.B.C + A'.B.C + A'.C.D
= A.B'.C'.D' + A.B.C'.D' + A.B.C.(D+D') + A'.B.C.(D+D') + A'.C.D.(B+B')
= A.B'.C'.D' + A.B.C'.D' + A.B.C.D + A.B.C.D' + A'.B.C.D + A'.B.C.D' + A'.B'.C.D



PIs and EPIs

To find the simplest (minimal) SOP expression from a K-map, you need to obtain:

- Minimum number of literals per product term; and
- Minimum number of product terms

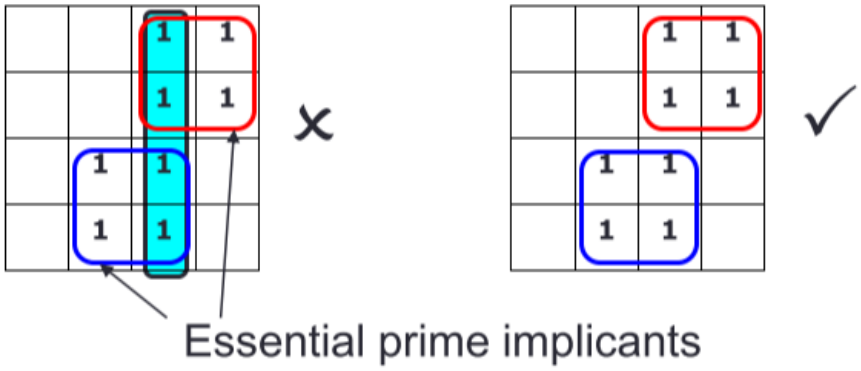
Achieved using K-map using

- Bigger groupings of minterms (prime implicants) where possible and
- No redundant groupings (looking for essential prime implicants)

Implicant: A product term that can be used to cover minterms of the function.

Prime Implicant (PI): A product term obtained by combining the maximum number of minterms from adjacent squares in the map.

Essential Prime Implicant (EPI): A prime implicant that includes atleast one minterm that is not covered by any other prime implicant.



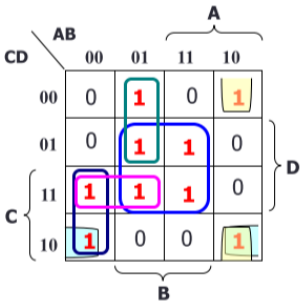
- No redundant groups

Finding Simplified SOP Expression

1. Circle all prime implicants on the K-map
2. Identify and select all EPI for the cover
3. Select a minimum subset of the remaining PI to complete the cover, that is, to cover those minterms not covered by the essential prime implicants

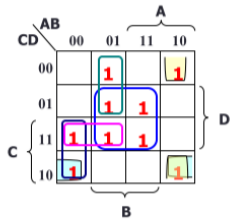
Example #1:

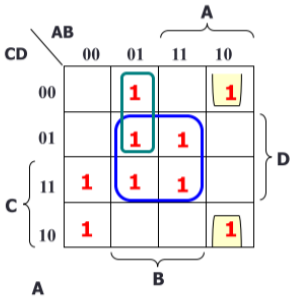
$$F(A,B,C,D) = \Sigma\, m(2,3,4,5,7,8,10,13,15)$$



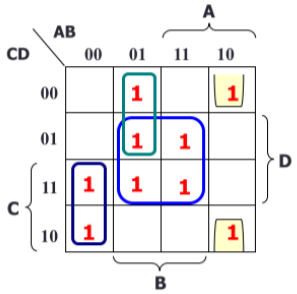
All prime implicants

All PIs





Essential prime implicants

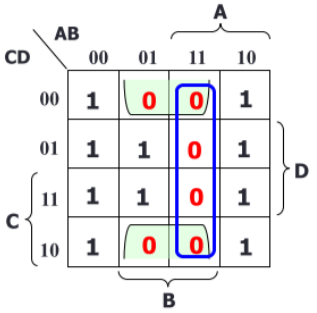


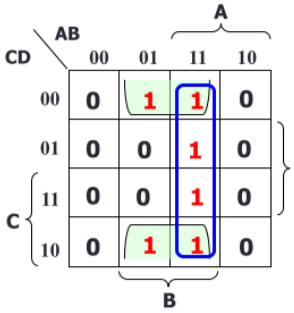
Answer

Finding Simplified POS Expression

Simplified POS expression can be obtained by grouping the maxterms (0s) of the given function.

K-map of F





K-map of F'

- This gives the SOP of F' to be

  $F' = B \cdot D' + A \cdot B$

- To get POS of F, we have

$$\begin{aligned} F &= (B \cdot D' + A \cdot B)' \\ &= (B \cdot D')' \cdot (A \cdot B)' && \text{(DeMorgan)} \\ &= \mathbf{(B'+D) \cdot (A'+B')} && \text{(DeMorgan)} \end{aligned}$$

Don't Care Conditions

Some outputs are not specified or not valid. These outputs can either be 1 or 0. These are called don't care conditions denoted by X.

Don't-care conditions can be used to simplify Boolean expression further in K-maps. They could be chosen to be either '1', or '0', depending on which choice results in simpler expression.

Example:

$F(A,B,C) = \Sigma m(3, 5, 6) + \Sigma d(0, 7)$

$G(A,B,C) = \Sigma m(1, 2, 4) + \Sigma d(0, 7)$

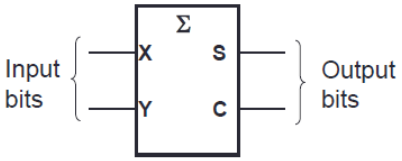
Assuming inputs 000 and 111 are invalid.

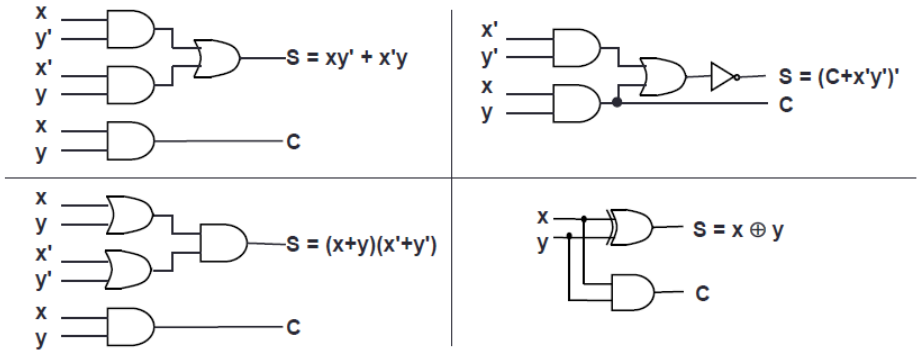
A	B	C	F	G
0	0	0	X	X
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	X	X

17. Combinational Circuits

Half adder

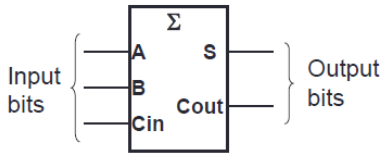
x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

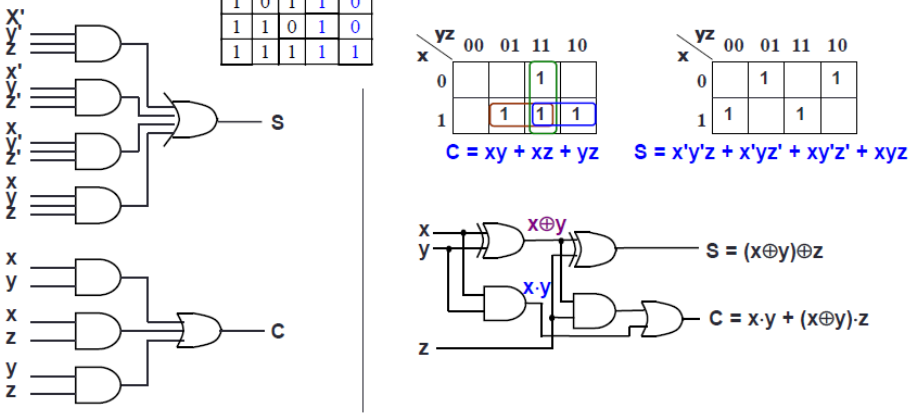




Full adder

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1





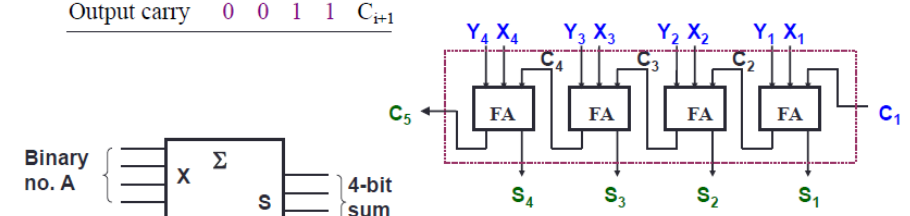
*Full Adder is made from two Half-Adders (and an additional OR gate)

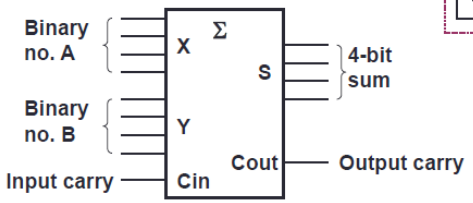
4-bit parallel adder

Subscript i	4	3	2	1	
Input carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}

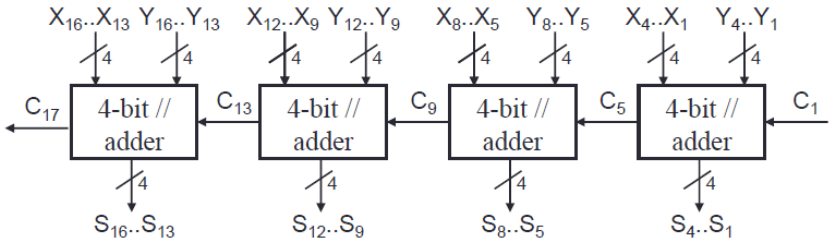
2 ways:

- Serial (one FA)
- Parallel (*n* FAs for *n* bits)



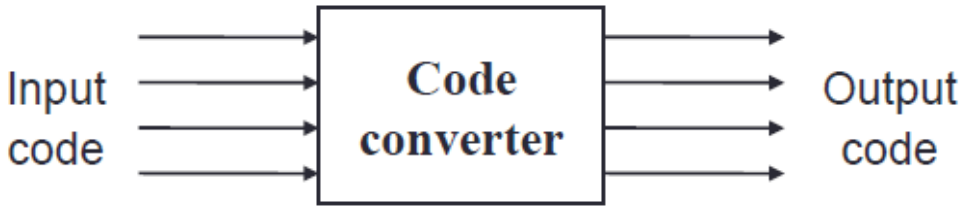


- Cascading 4 full adders (FAs) gives a 4-bit parallel adder.
  - Classical method: 9 input variables → 2^9 = 512 rows in truth table!
- Cascading method can be extended to larger adders.
  - Example: 16-bit parallel adder.



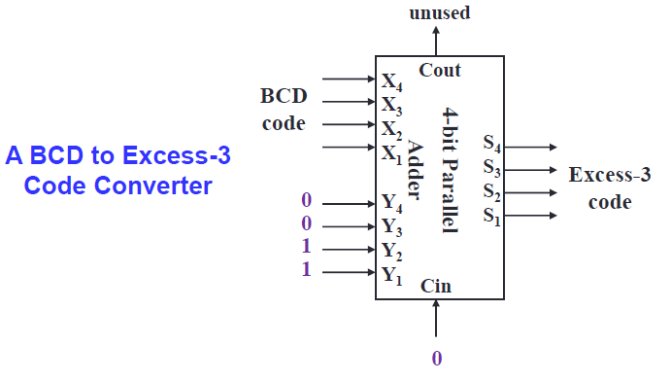
Code Converter

Takes an input code, translates to it equivalent output code.

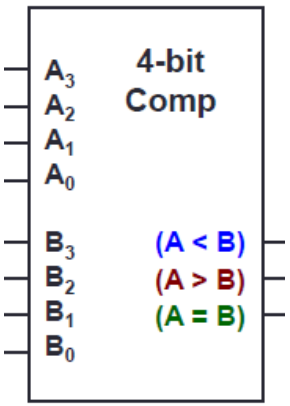


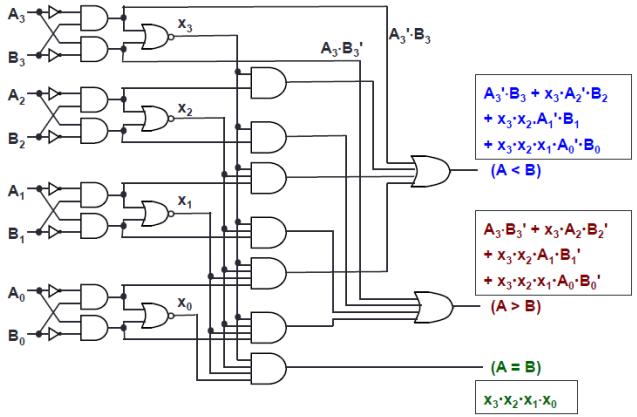
Example: BCD to Excess-3 code Converter

Excess-3 Code = BCD Code + 0011_2

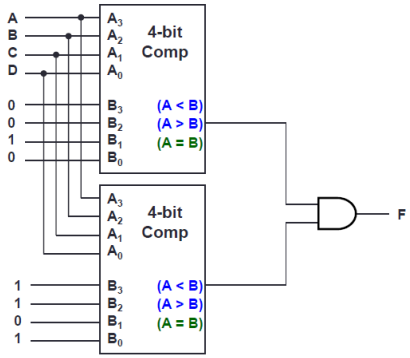


Magnitude Comparator

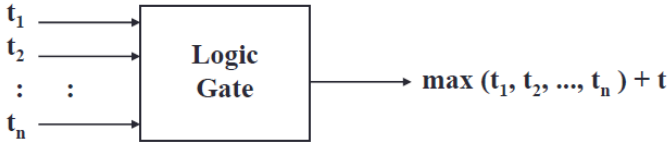




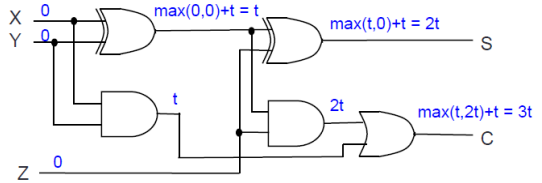
F(ABCD) = 1 IF $3 \leq ABCD \leq 12$



Circuit Delays



Given a logic gate with delay t. If inputs are stable at times $t_1, t_2, \dots, t_n$, then the earliest time in which the output will be stable is $\mathbf{max}(\boldsymbol{t_1}, \boldsymbol{t_2}, \dots, \boldsymbol{t_n}) + \boldsymbol{t}$



Output S and S experience delays of 2t and 3t respectively.

In general, an n-bit ripple carry // adder will experience

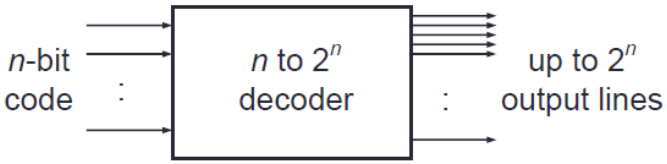
$S_n = \big((n-1)2 + 2\big)t$

$C_{n+1} = \big((n-1)2 + 3\big)t$ (maximum delay for ripple carry // adder)

Propagation delay of ripple carry // adders is proportional to the number of bits it handles.

18. MSI Components

Decoders

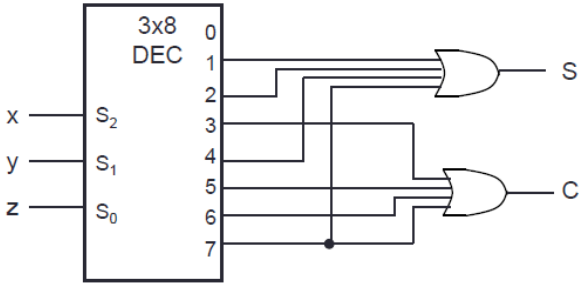


In general, for n-bit code, a decoder could select up to 2^n lines.

Implementing Functions (Decoder)

- A boolean function in SOM form
  - Decoder to generate the minterms and an OR gate to form the sum
- Any combinational circuit with n inputs and m outputs can be implemented with an n: 2^n decoder with m OR gates
- Good when circuit has many outputs, and each function is expressed with a few minterms.

Example: Full Adder



E	X	Y	F_0	F_1	F_2	F_3
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

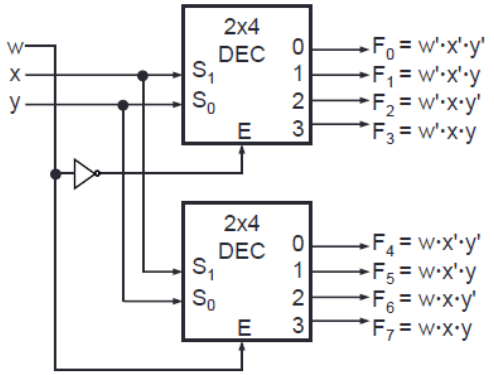
Decoder with 1-enable

E'	X	Y	F_0	F_1	F_2	F_3
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

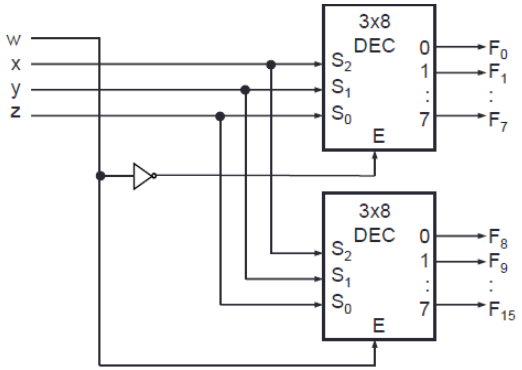
Decoder with 0-enable

Constructing Larger Decoders

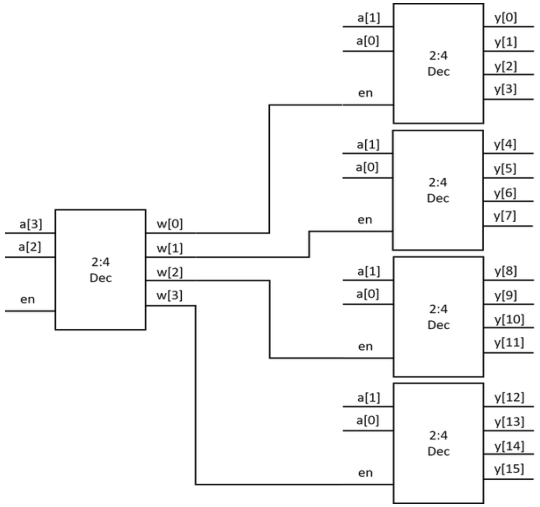
Larger decoders can be constructed from smaller ones.



3 × 8 decoder can be built using two 2 × 4 decoders (with one-enable) and an inverter.



4 × 16 decoder using two 3 × 8 decoders with one-enable and an inverter.



Constructing 4 × 16 decoder using five 2 × 4 decoders with enable.

Standard MSI Decoder

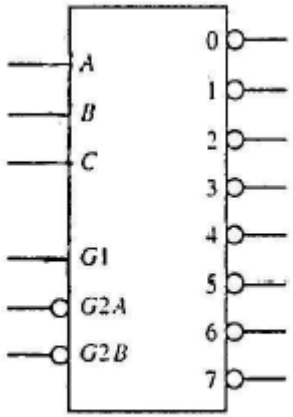
74138 (3 to 8 decoder)

INPUTS					OUTPUTS							
ENABLE		SELECT										
G1	$\bar{G}2^*$	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	H	H	H	H	H	L	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	L	H	H	H	H	H	H	L	H	H
H	L	H	H	L	H	H	H	H	H	H	L	H
H	L	H	H	H	H	H	H	H	H	H	H	L

$^*\bar{G}2 = \bar{G}2A + \bar{G}2B$

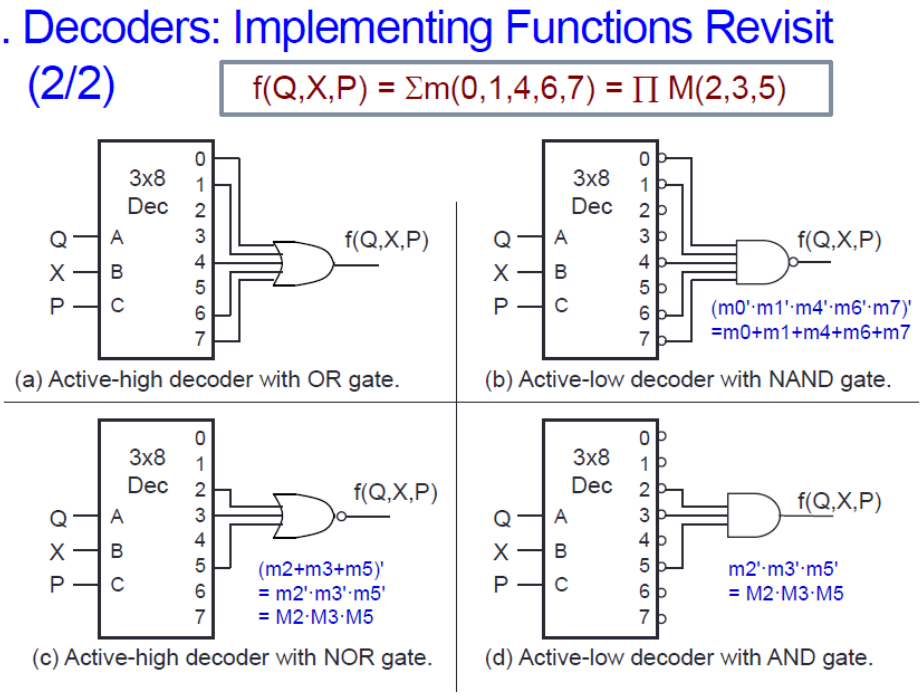
H = high level, L = low level, X = irrelevant

(c)



(d)

Implementing Functions

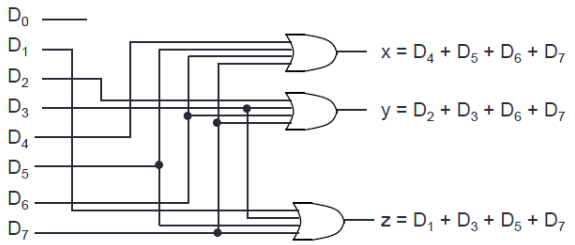


Encoders

Contains 2^n (or fewer) input lines and n output lines.

Example: 8 to 3 encoder

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1



Priority Encoder

A priority encoder is one with priority. If two or more inputs are equal to 1, the input with highest priority takes precedence.

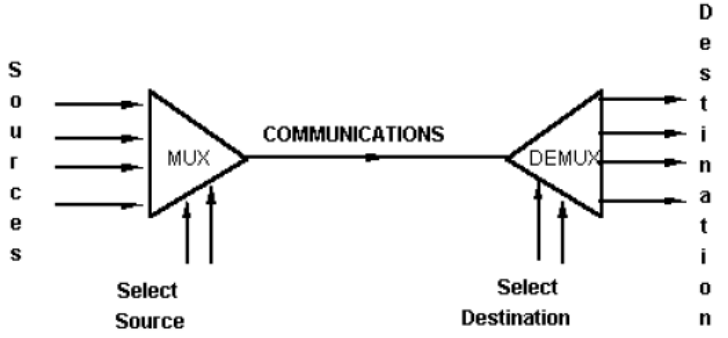
If all inputs are 0, this input combination is considered invalid.

Example: 4 to 2 priority encoder

Inputs				Outputs		
D_0	D_1	D_2	D_3	*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

Multiplexers and Demultiplexers

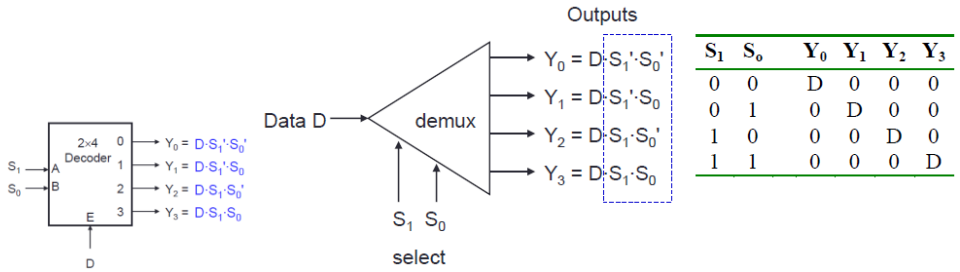
Helps to share a single communication line among a number of devices.



Demutiplexers

Given an input line and a set of selection lines, a demux directs data from the input to one selected output line.

A demultiplexer circuit is actually identical to a decoder with enable.



S_1	S_o	Y_0	Y_1	Y_2	Y_3
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D

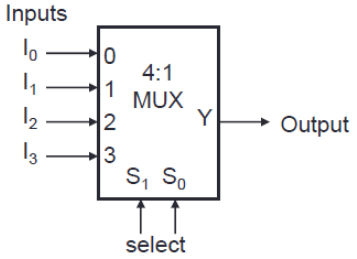
Multiplexers

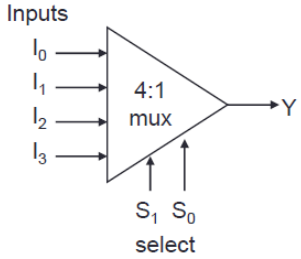
A multiplexer is a device that has a number of input lines, a number of sleection lines, one output lines.

It steers one of 2^n inputs to a single output line, using n selection lines. Also known as data selector.

I_0	I_1	I_2	I_3	S_1	S_0	Y
d_0	d_1	d_2	d_3	0	0	d_0
d_0	d_1	d_2	d_3	0	1	d_1
d_0	d_1	d_2	d_3	1	0	d_2
d_0	d_1	d_2	d_3	1	1	d_3

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3



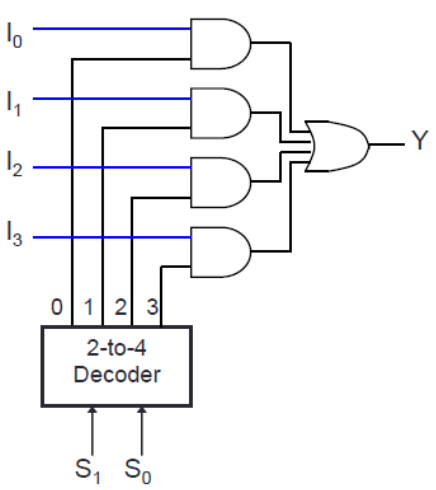


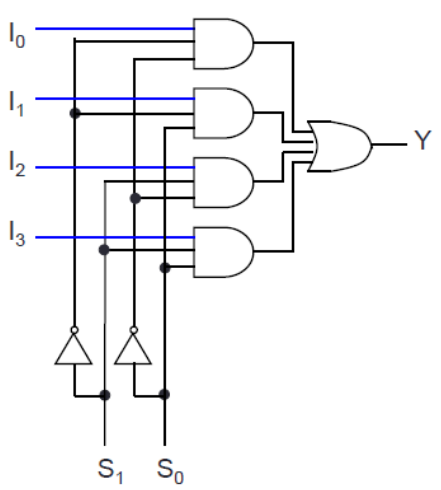
Example: Output of a 4-to-1 multiplexer is:

$$Y = I_0 \cdot (S_1' \cdot S_0') + I_1 \cdot (S_1' \cdot S_0) + I_2 \cdot (S_1 \cdot S_0') + I_3 \cdot (S_1 \cdot S_0)$$

A 2^n to 1 line mutliplexer, or simply 2^n:1 MUX, is made from an n: 2^n decoder by adding to it 2^n input lines, one to each AND gate.

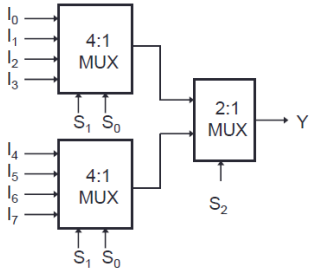
4:1 MUX Circuit



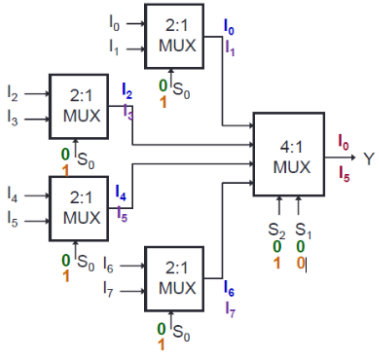


Constructing Larger Multiplexers

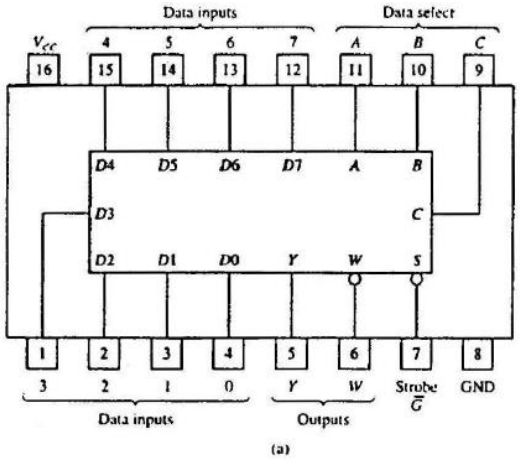
8:1 MUX



S_2	S_1	S_0	Y
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7



Standard MSI Multiplexer



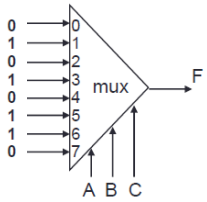
(a)

INPUTS				OUTPUTS	
SELECT			STROBE		
C	B	A	$\overline{G}$	Y	W
X	X	X	H	L	H
L	L	L	L	D0	$\overline{D0}$
L	L	H	L	D1	$\overline{D1}$
L	H	L	L	D2	$\overline{D2}$
L	H	H	L	D3	$\overline{D3}$
H	L	L	L	D4	$\overline{D4}$
H	L	H	L	D5	$\overline{D5}$
H	H	L	L	D6	$\overline{D6}$
H	H	H	L	D7	$\overline{D7}$

(b)

Implementing Functions (Multiplexer)

- F(A,B,C) = Σ m(1,3,5,6)



This method works because:

Output = $I_0 \cdot m_0 + I_1 \cdot m_1 + I_2 \cdot m_2 + I_3 \cdot m_3 + I_4 \cdot m_4 + I_5 \cdot m_5 + I_6 \cdot m_6 + I_7 \cdot m_7$

Supplying '1' to I_1, I_3, I_5, I_6 , and '0' to the rest:

Output = $m_1 + m_3 + m_5 + m_6$

2^n to 1 MUX can be used to implement a Boolean function of n variables. However, we can use a single smaller 2^{n-1} to 1 MUX to implement a Boolean function of n variables.

F(A,B,C) can be implemented using a 4 to 1 MUX rather than an 8 to 1 MUX.

19. Sequential Logic

Flip Flop Characteristic Table

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	Reset
1	1	Set

T	Q(t+1)	
0	Q(t)	No change
1	Q(t)'	Toggle

Excitation Tables

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

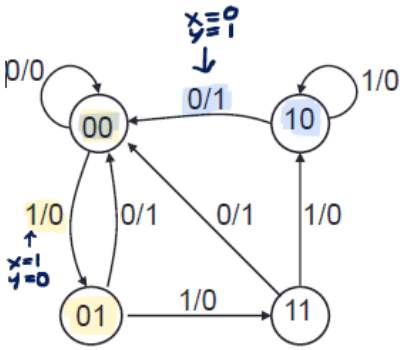
T Flip-flop

State Diagram

- Each state denoted by a circle
- Each arrow denotes a transition of the sequential circuit
- A label of the form a/b is attached to each arrow where a (if there is one) denotes the input while b (if there is one) denotes the outputs of the circuits in that transition.
- Each combination of the flip-flop values represents a state. Hence, m flipflops → up to 2^m states.

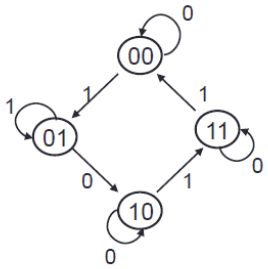
Present State	Next State x=0	Next State x=1	Output x=0	Output x=1
AB	A^+B^+	A^+B^+	y	y
00	00	01	0	0
01	00	11	1	0
10	00	10	1	0
11	00	10	1	0

input / output



DONE!

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



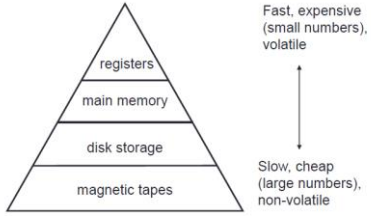
Q	Q^+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

JK Flip-flop's excitation table.

Present State	Next State x=0	Next State x=1
AB	A^+B^+	A^+B^+
00	00	01
01	10	01
10	10	11
11	11	00

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

Memory



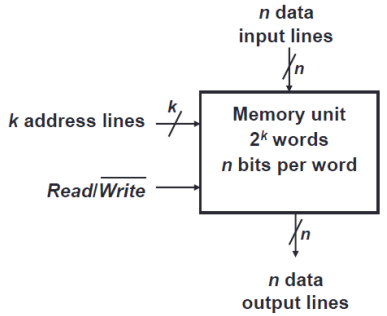
Definitions:

1 byte = 8 bits

1 word = multiple of bytes

1 KB = 2^{10} bytes; 1MB = 2^{20} bytes; 1GB = 2^{30} bytes …

Desirable Properties: Fast access, large capacity, economical cost, non-volatile



Memory Enable	*Read/Write*	Memory Operation
0	X	None
1	0	Write to selected word
1	1	Read from selected word

20-21. Pipelining

- Pipelining doesn't help the latency of single task, it helps the throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Possible Delays:
  - Pipeline rate limited by slowest pipeline stage
  - Stall for dependencies

MIPS Pipeline Stages

Five Execution Stages

- **IF**: Instruction Fetch
- **ID**: Instruction Decode and Register Read
- **EX**: Execute an operation or calculate an address
- **MEM**: Access an operand in data memory
- **WB**: Write back the result into a register

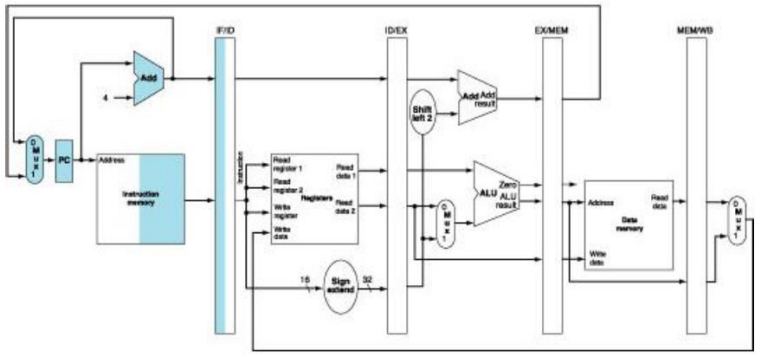
Each execution stage takes 1 clock cycle. General flow of data is from one stage to the next.

Exceptions: Update of PC and WB of register file

MIPS Pipeline: Datapath

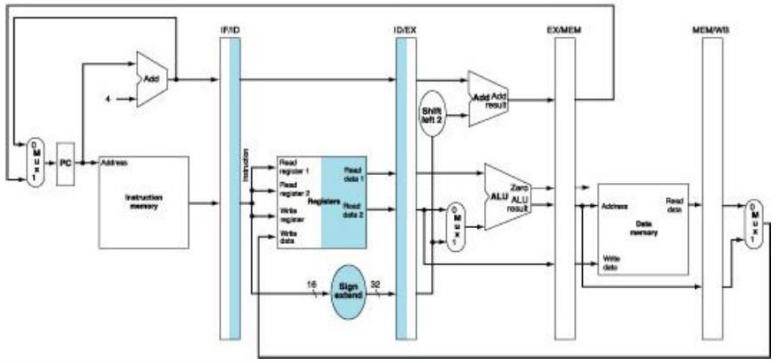
- Single-cycle implementation
  - Update all state elements (PC, RegFile, DataMem) at the end of a clock cycle
- Pipelined implementation
  - One cycle per pipeline stage
  - Data required for each stage needs to be stored separately
- Data used by subsequent instructions:
  - Store in programmer-visible state elements: PC, RegFile, Mem
- Data used by same instruction in later pipeline stages:
  - Additional registers in datapath called **pipeline registers**
  - IF/ID: Register between IF and ID
  - ID/EX
  - EX/MEM
  - MEM/WB

3. Pipeline Datapath: `IF` Stage



- At the end of a cycle, `IF/ID` receives (stores):
  - Instruction read from InstructionMemory[PC]
  - PC + 4
- PC + 4
  - Also connected to one of the MUX's inputs (another coming later)

3. Pipeline Datapath: `ID` Stage



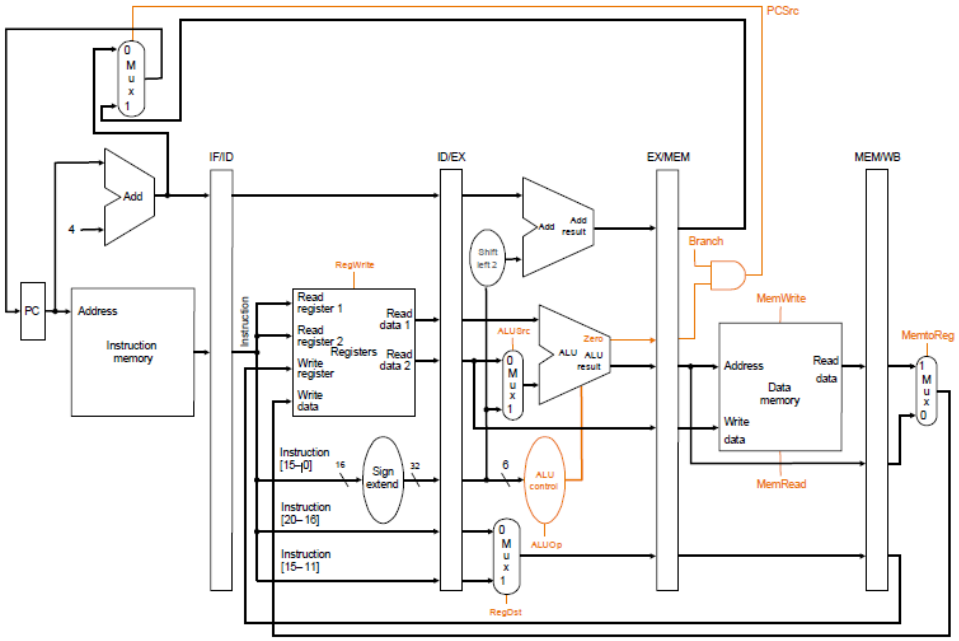
At the beginning of a cycle `IF/ID` register supplies:	At the end of a cycle `ID/EX` receives:
❖ Register numbers for reading two registers ❖ 16-bit offset to be sign-extended to 32-bit ❖ `PC + 4`	❖ Data values read from register file ❖ 32-bit immediate value ❖ `PC + 4`

Corrected Datapath

"Write register" number supplied by the IF/ID pipeline register is INCORRECT. It is not the correct write register for the instruction now in WB stage.

Solution:

- Pass the "Write register" number form ID/EX through EX/MEM to MEM/WB pipeline register for use in WB stage.
- Let the "WR" number follow the instruction through pipeline until it is needed in the WB stage.



Pipeline Control: Main Idea

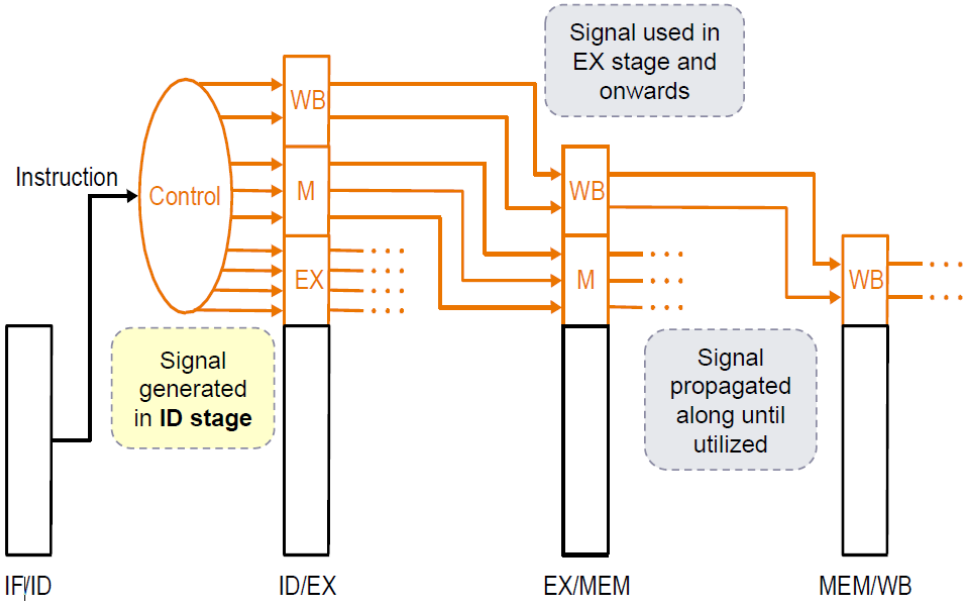
- Same control signals as single-cycle datapath
- Difference: Each control signal belongs to a particular pipeline stage

Pipeline Control: Grouping

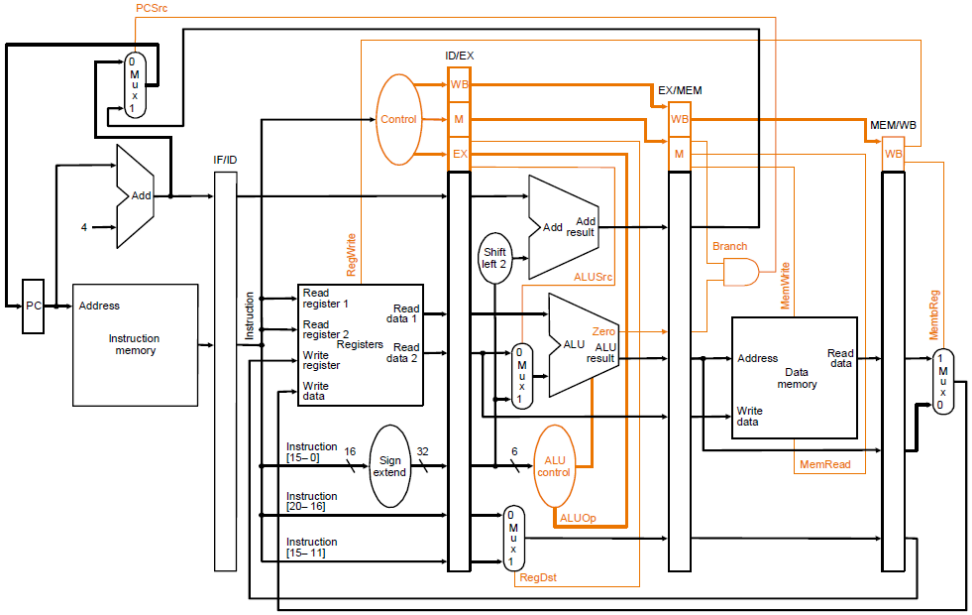
	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

	EX Stage				MEM Stage			WB Stage	
	RegDst	ALUSrc	ALUop		Mem Read	Mem Write	Branch	MemTo Reg	Reg Write
			op1	op0					
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

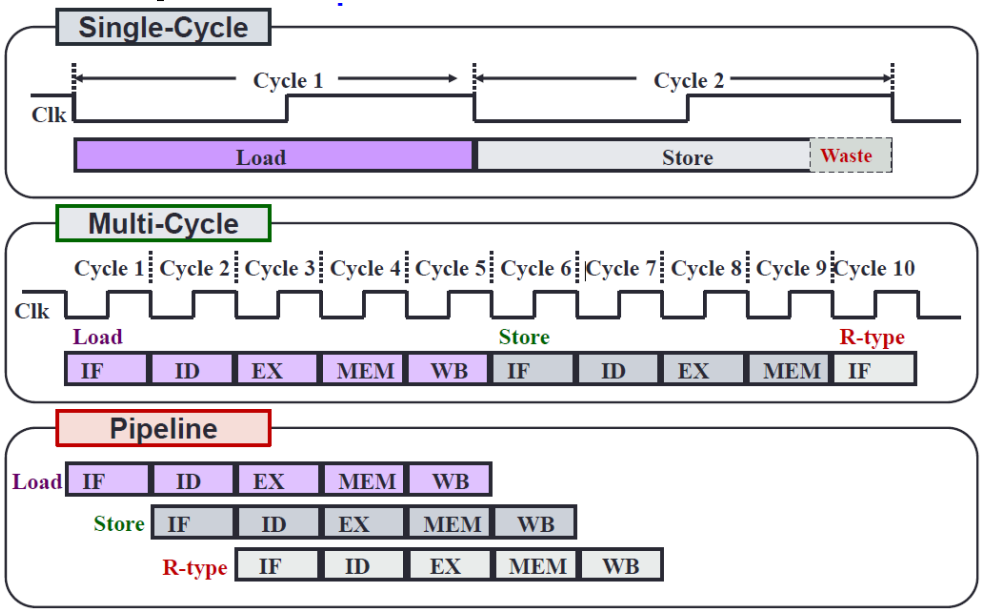
Pipeline Control: Implementation



Pipeline Control: Datapath and Control



Different Implementations

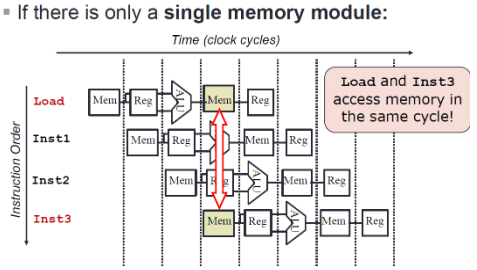


Pipeline Hazards

Problems that prevent next instruction from immediately following previous instruction

Structural Hazards: Simultaenous use of a hardware resource
Data Hazards: Data dependencies between instructions
Control Hazards: Change in program flow

Structural Hazards:



Solution 1: Stall pipeline (1 cycle)

Solution2:
Split memory into Data and Instruction memory
Split cycle into half, first half for writing and second half for reading

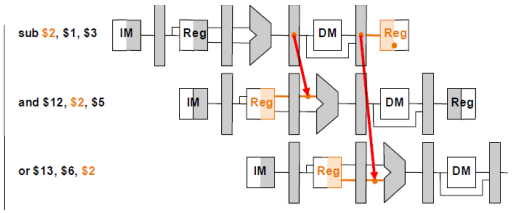
Data Dependency

RAW: Read-After-Write
Occurs when a later instruction reads from the destination register written by an earlier instruction
Also known as true data dependency

WAR/WAW: Write-After-Read, Write-after-Write
These dependencies do not cause any pipeline hazards.

Solution:

- Forward the result to any trailing instructions before it is reflected in register file
- Bypass (replace) the data read from register file



Data Hazards: Forwarding

Without forwarding (Load/ No Load)

	1	2	3	4	5	6	7	8	9	10	11
sub	IF	ID	EX	MEM	WB						
and		IF			ID	EX	MEM	WB			
or					IF	ID	EX	MEM	WB		

With Forwarding (No Load)

	1	2	3	4	5	6	7	8	9	10	11
sub	IF	ID	EX	MEM	WB						
and		IF	ID	EX	MEM	WB					
or			IF	ID	EX	MEM	WB				

With Forwarding (Load)

	1	2	3	4	5	6	7	8	9	10	11
lw	IF	ID	EX	MEM	WB						
and		IF	ID		EX	MEM	WB				
or			IF		ID	EX	MEM	WB			

Control Hazard

Branching is common in code – 3-cycle stall is too heavy (need to wait until MEM stage).

Early Branch Resolution (Make decision in ID instead of MEM)

Move branch target address calculation, move register comprison; cannot use ALU for register comparison anymore

Branch Prediction

Simple Prediction: Assume branches NOT TAKEN.

When actual branch not taken -> no pipeline stall

When actual branch taken -> Flush successor instruction from pipeline

Without Branch Prediction

	1	2	3	4	5	6	7	8	9	10	11
addi[1]	IF	ID	EX	MEM	WB						
addi[2]		IF	ID	EX	MEM	WB					
bne			IF		ID	EX	MEM	WB			
addi[2]						IF	ID	EX	MEM	WB	

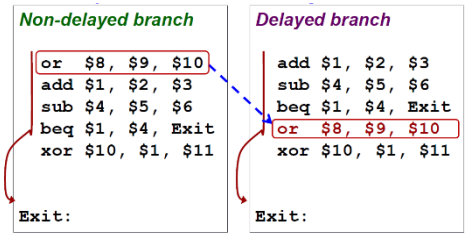
With Branch Prediction

	1	2	3	4	5	6	7	8	9	10	11
addi[1]	IF	ID	EX	MEM	WB						
addi[2]		IF	ID	EX	MEM	WB					
bne			IF		ID	EX	MEM	WB			
sub					IF						
addi[2]						IF	ID	EX	MEM	WB	

Delayed Branching/Prediction

Branch outcome takes X number of cycles to be known -> X cycle stalls

Idea: Move non-control dependent instructions into the X slots following a branch (known as branch-delay slot). These instructions are executed regardless of the branch outcome



Best case scenario: There is an instruction preceding the branch which can be moved into the delayed slot

Worst case scenario : Such instruction cannot be found

22. Cache: Direct Mapped

Key Concept:

Use a hierachy of memory technologies:

- Small but fast memory near CPU
- Large but slow memory farther away from CPU

Cache: The Basic Idea

Keep the frequently and recently used data in smaller but faster memory.
Refer to bigger and slower memory only when you cannot find data/instruction in the faster memory.

Principle of Locality

Program accesses only a small portion of the memory address space within a small time interval.

Types of Locality

Temporal Locality

If an item is referenced, it will tend to be reference again soon.

Spatial Locality

If an item is referenced, nearby items will tend to be referenced soon.

Different locality for

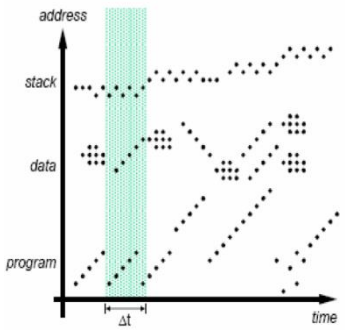
- Instructions
- Data

Definition: Working Set

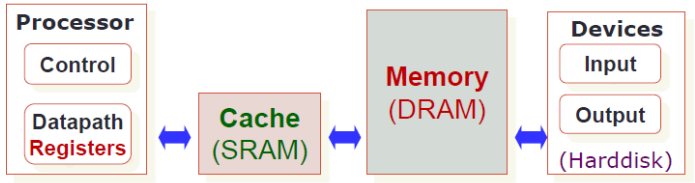
Set of locations accessed during Δt

* Different phases of execution may use different working sets

Our aim is to capture the working set and keep it in the memory closest to CPU.



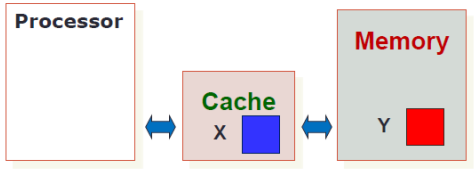
Two Aspects of Memory Access



How to make SLOW main memory appear FASTER?
Cache – a small but fast SRAM near CPU
Hardware managed: Transparent to programmer

How to make SMALL main memory appear BIGGER?
Virtual Memory
OS Managed: Transparent to programmer

Memory Acccess Time



Hit: Data is in cache (e.g. X)

Hit Rate: Fraction of memory accesses that hit
Hit Time: Time to access cache

Miss: Data is not in cache (e.g. Y)

Miss rate = $1 - Hit\ rate$
Miss penalty: Time to replace cache block + hit time

Hit Time < Miss Penalty

Average Access Time

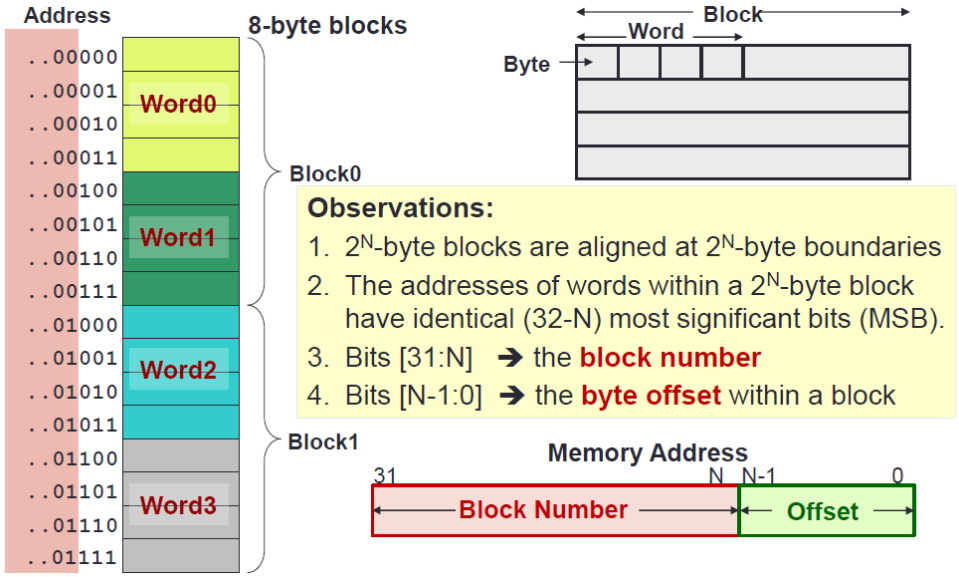
$= Hit\ Rate \times Hit\ Time + (1 - Hit\ Rate) \times Miss\ Penalty$

Memory to Cache Mapping

Cache Block/Line:
Unit of transfer between memory and cache

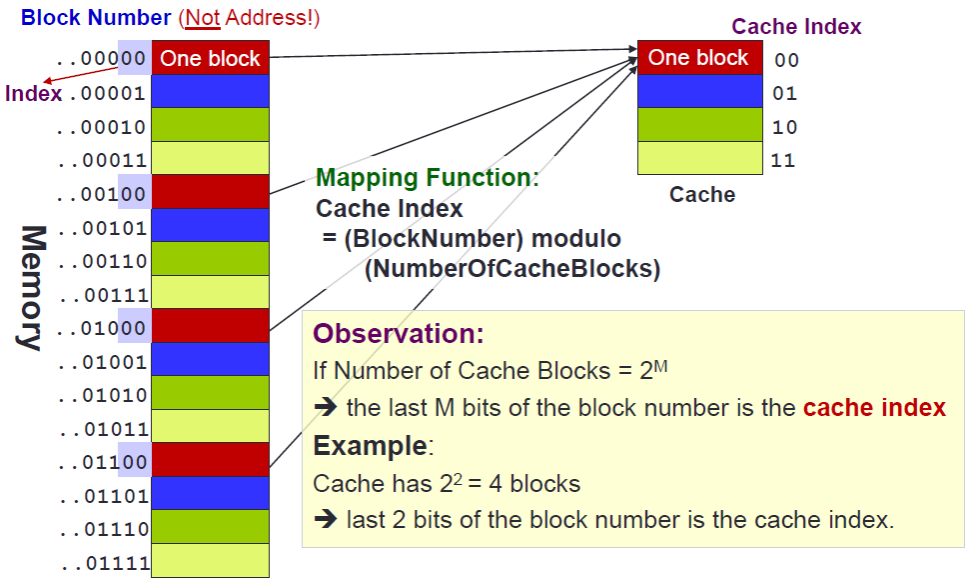
Block size is typically one or more words
- 16-byte block = 4-word block
- 32-byte block = 8-word block

The block size is bigger than word size to exploit locality.

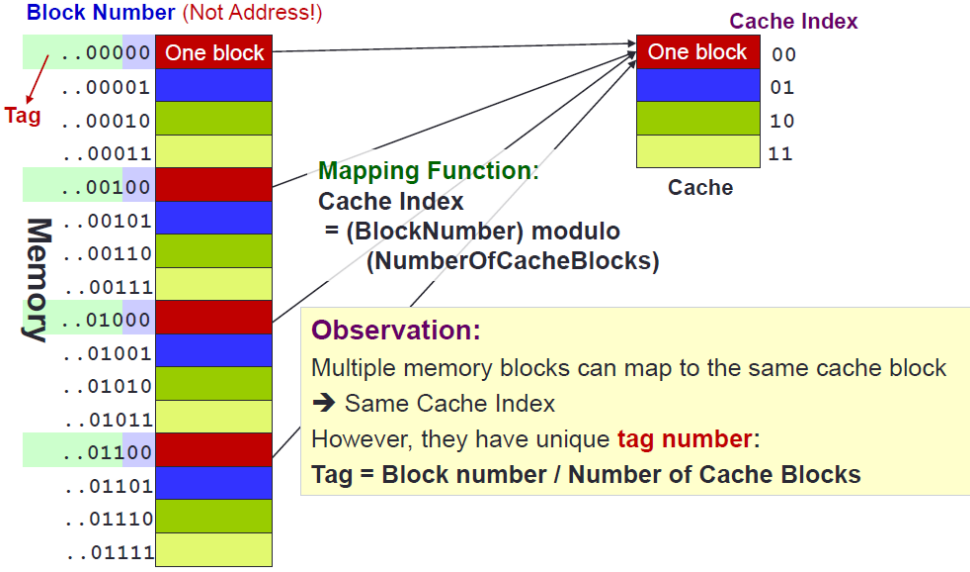


*Assume 1 word contains 4 bytes
*N = 3

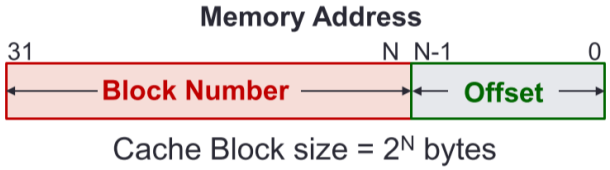
Direct Mapped Cache: Cache Index

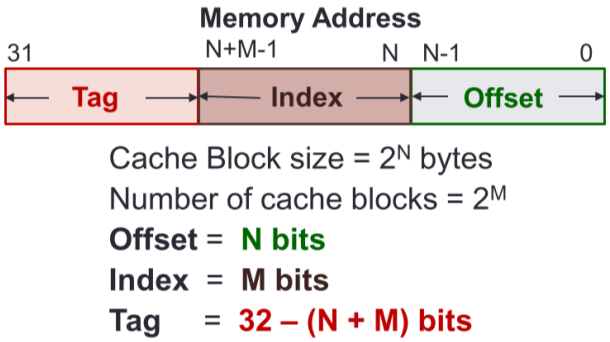


Direct Mapped Cache: Cache Tag



Direct Mapped Cache: Mapping





Types of Cache Miss

Compulsory Misses

- On the first access to a block; the block must be brought into the cache
- Also called *cold start misses* or *first reference misses*

Conflict misses

- Occur in the case of direct mapped cache or set associative cache, when several blocks are mapped to the same block/set
- Also called *collision misses* or *interference misses*

Capacity misses

- Occur when blocks are discarded from cache as cache cannot contain all blocks needed

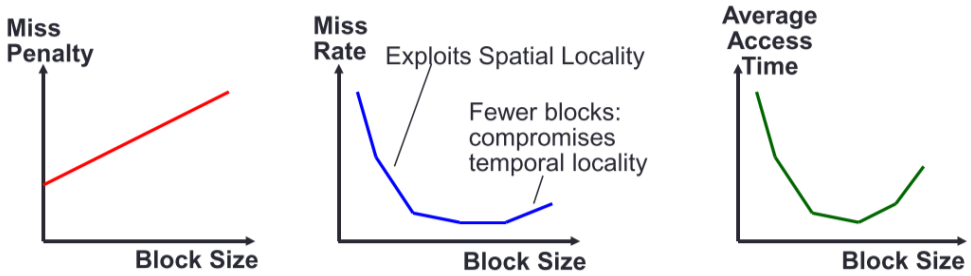
23. CACHE: Set/Fully Associative

Block Size Trade-off

Average Access Time = Hit Rate * Hit Time + (1 – Hit Rate) * Miss Penalty

Larger Block Size:

- Takes advantage of spatial locality
- Larger miss penalty: Takes longer to fill up the blocok
- If the block size it too big relative to cache size
    - Too few cache blocks; miss rate will go up



Set Associative (SA) Cache

Conflict Misses: Occur in the case of direct mapped cache or set associative cache, when several blocks are mapped to the same block/set.

Solution: Set Associative Cache

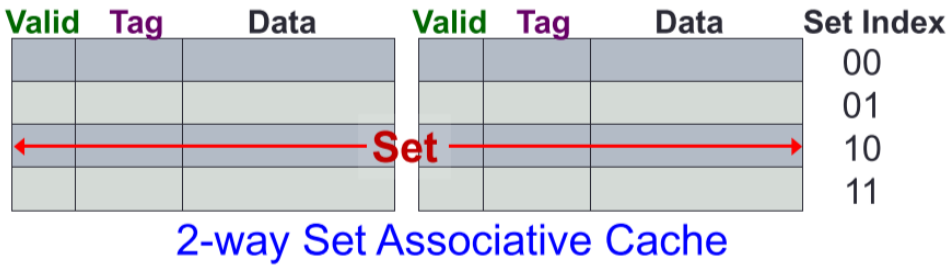
N-way Set Associative Cache

A memory block can be placed in a fixed number (N) of locations in the cache, where N > 1

Key Idea:

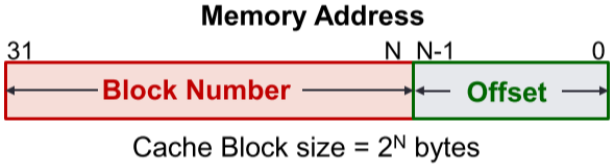
- Cache consists of a number of sets: Each set contains N cache blocks
- Each memory block maps to a unique cache set
- Within the set, a memory block can be placed in any of the N cache blocks in the set

Example: 2-way set associative cache



- Each set has two cache blocks
- A memory block maps to a **unique set**
  - In the set, the memory block can be placed in **either of the cache blocks**
  - Need to search both to look for the memory block

Set Associative Cache: Mapping



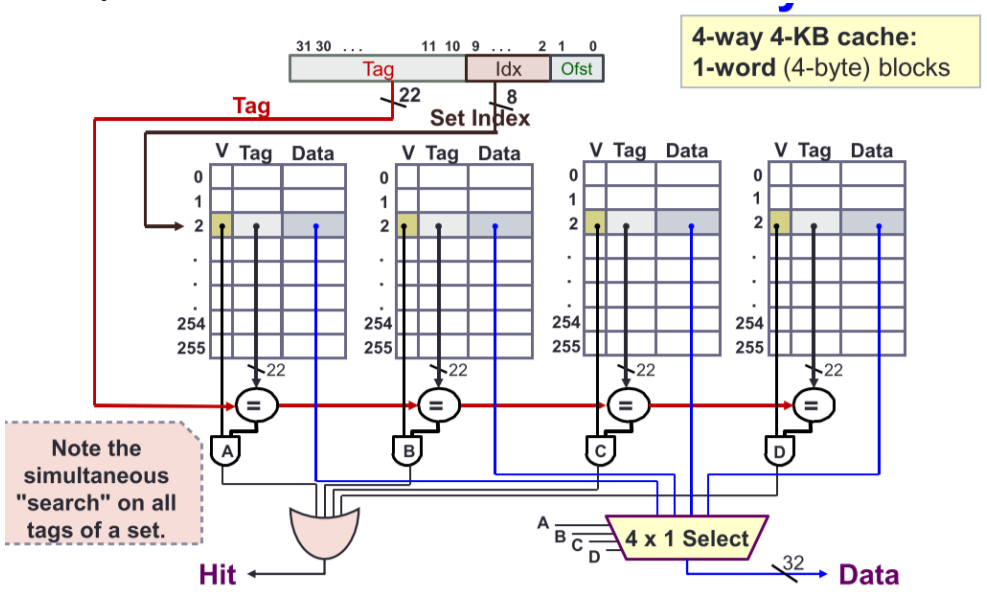
Cache Block size = 2^N bytes

Cache Set Index
= (BlockNumber) modulo (NumberOfCacheSets)

Cache Block size = 2^N bytes
Number of cache sets = 2^M
Offset = **N bits**
Set Index = **M bits**
Tag = **32 – (N + M) bits**

Observation:
It is essentially unchanged from the direct-mapping formula

Circuitry



Advantage of Associativity

Rule of Thumb:

A direct-mapped cache of size N has about the same miss rate as a 2-way set associative cache of size N/2.

Fully Associative (FA) Cache

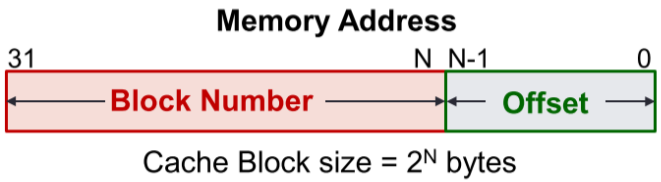
Capacity misses: Occur when blocks are discarded from cache as cache cannot contain all blocks needed (Occurs in FA Cache)

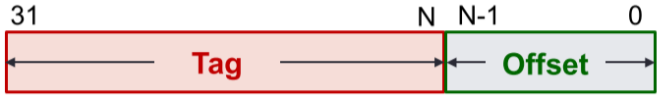
A memory block can be placed in any location in the cache.

Key Idea:

- Memory block placement is no longer restricted by cache index or cache set index
- ++ Can be placed in any location, BUT
- --- Need to search all cache blocks for memory access

FA Cache: Mapping





Cache Block size = 2^N bytes
Number of cache blocks = 2^M
Offset = **N bits**
Tag = **32 – N bits**

Observation:
The block number serves as the tag in FA cache.

Block Replacement Policy

Set Associative or Fully Associative Cache:

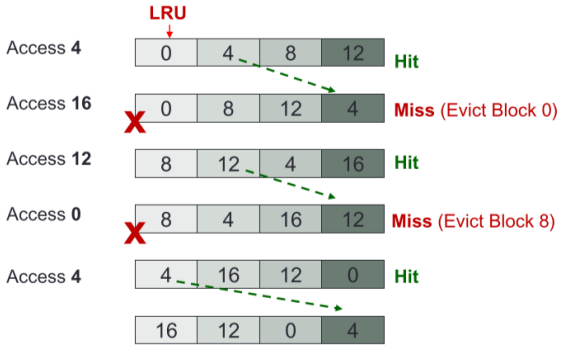
- Can choose where to place a memory block
- Potentially replacing another cache block if full
- Need block replacement policy

Least Recently Used (LRU)

How: For cache hit, record the cache block that was accessed. When replacing a block, choose one which has not been accessed for the longest time.

Why: Temporal Locality

- Least Recently Used policy in action:
  - 4-way SA cache
  - Memory accesses: **0 4 8 12 4 16 12 0 4**



Drawback for LRU

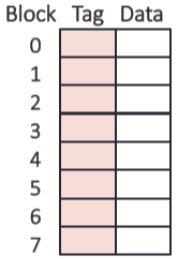
- Hard to keep track if there are many choices

Other replacement policies:

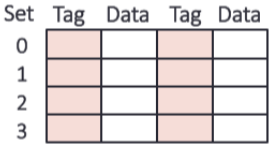
- FIFO
- Random replacement (RR)
- Least frequently used (LFU)

Summary

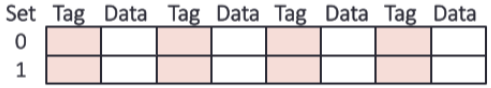
One-way set associative (direct mapped)



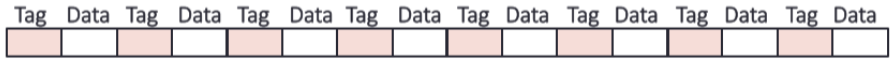
Two-way set associative



Four-way set associative



Eight-way set associative (fully associative)



Block Placement: Where can a block be placed in cache?

Direct Mapped:	N-way Set-Associative:	Fully Associative:
• Only one block defined by index	• Any one of the **N** blocks within the set defined by index	• Any cache block

Block Identification: How is a block found if it is in the cache?

Direct Mapped:	N-way Set Associative:	Fully Associative:
• Tag match with only one block	• Tag match for all the blocks within the set	• Tag match for all the blocks within the cache

Block Replacement: Which block should be replaced on a cache miss?

Direct Mapped:	N-way Set-Associative:	Fully Associative:
• No Choice	• Based on replacement policy	• Based on replacement policy

Write Strategy: What happens on a write?

Write Policy: Write-through vs write-back

Write Miss Policy: Write allocate vs write no allocate