EGC442 Class Notes 5/9/2023

Baback Izadi
Division of Engineering Programs
bai@engr.newpaltz.edu

Final:

Comprehensive

- Performance problems
- ALU design
- Data Path and control
- Pipelining design and hazard ✓
- Cache memory
- Virtual memory
- Parallel Computing ✓

1. review notes

zybook

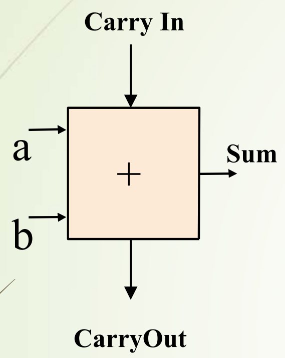
2. redo quiz

3. redo tests

4. HW's

Making a faster adder Full Adder

Let's look at a 1-bit ALU for addition:



$\text{Sum} = a \oplus b \oplus c_{in}$

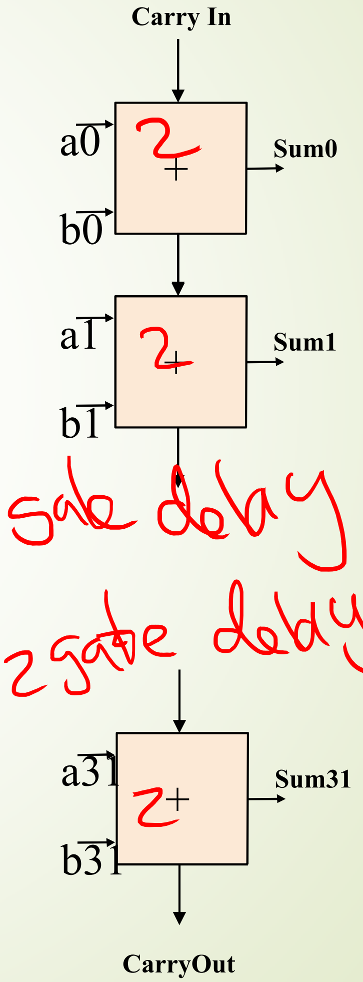
$C_{out} = a\,b + (a \oplus b)\,c_{in}$

$C_{out} = a\,b + a\,c_{in} + b\,c_{in}$

3 gate delay

2 gate delay

What is the propagation delay of a 32-bit adder?



32*2=64

Problem with Ripple Carry

- Is a 32-bit ALU as fast as a 1-bit ALU?
- Is there more than one way to do addition?
  - two extremes: ripple carry and sum-of-products
- Can you see the ripple? How could you get rid of it?

$c_1 = b_0c_0 + a_0c_0 + a_0b_0$

$c_2 = b_1c_1 + a_1c_1 + a_1b_1$ $c_2 =$

$c_3 = b_2c_2 + a_2c_2 + a_2b_2$ $c_3 =$

$c_4 = b_3c_3 + a_3c_3 + a_3b_3$ $c_4 =$

Not feasible! Why?

Carry-lookahead adder

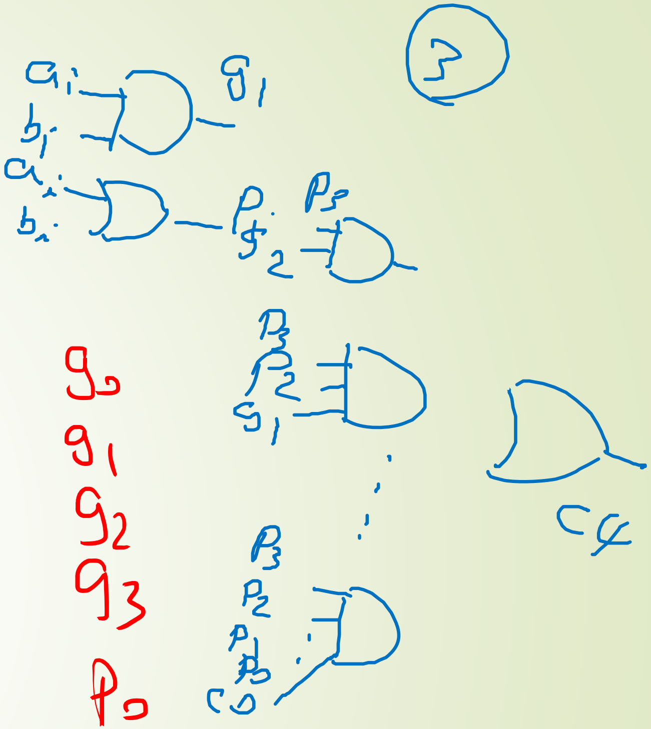
- An approach in-between our two extremes
- $c_1 = b_0c_0 + a_0c_0 + a_0b_0 = (b_0 + a_0)c_0 + a_0b_0$
  - If we didn't know the value of carry-in, what could we do?
  - When would we always generate a carry? $g_i = a_i b_i$ → 1
  - When would we propagate the carry? $p_i = a_i + b_i$ → 1
- Did we get rid of the ripple?

$c_1 = g_0 + p_0c_0$

$c_2 = g_1 + p_1c_1$ $\qquad c_2 = g_1 + P_1 g_0 + P_1 P_0 c_0$

$c_3 = g_2 + p_2c_2$ $\qquad c_3 = g_2 + c_2(\quad)$

$c_4 = g_3 + p_3c_3$ $\qquad c_4 = g_3 + P_3(\quad)$

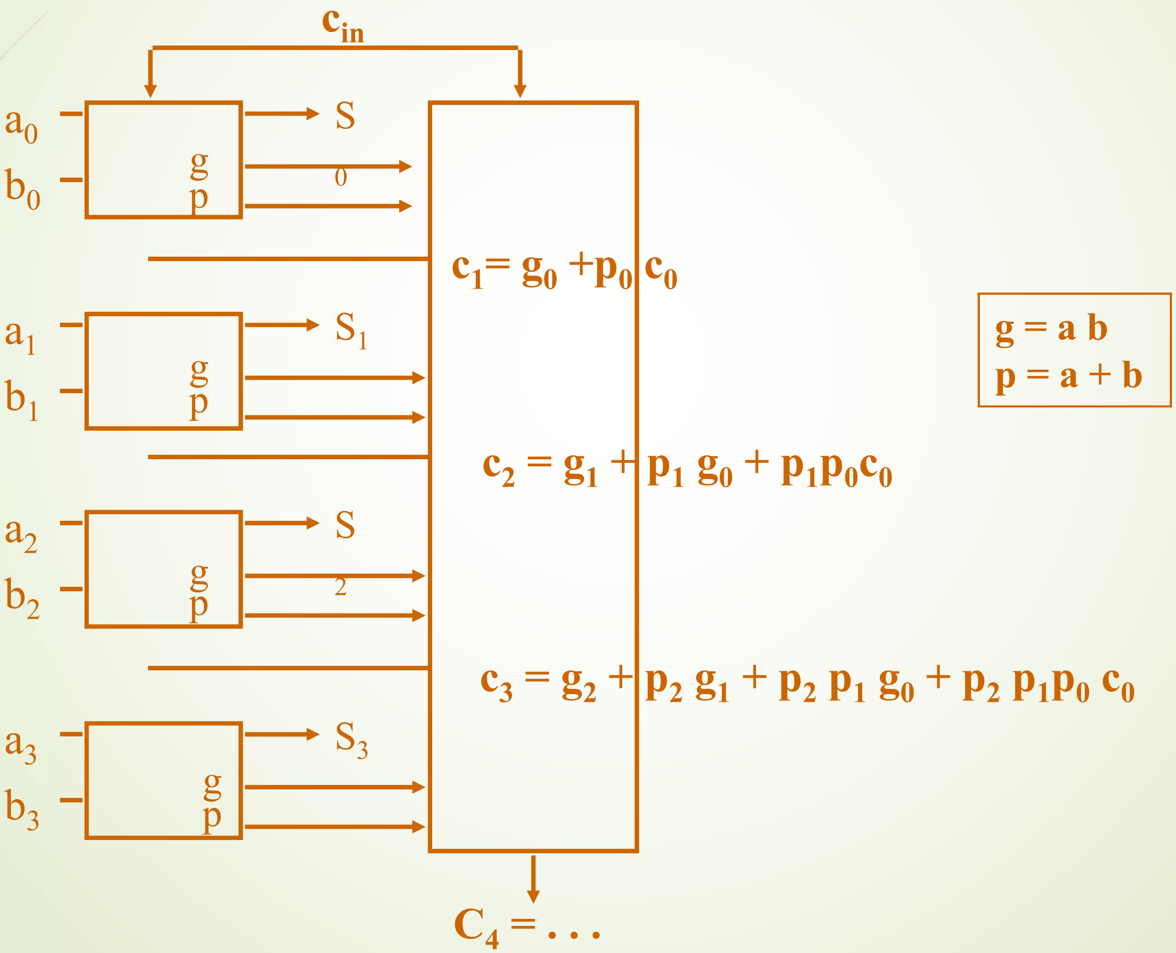


g_0
g_1
g_2
g_3
P_0
P_1
P_2
P_3

C_4

$c_4 = g_3 + p_3g_2 + p_3p_2\,g_1 + p_3\,p_2\,p_1\,g_0 + p_3\,p_2\,p_1p_0\,c_0$

Carry Look Ahead Design trick



$c_1 = g_0 + p_0 c_0$

$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$

$c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$

$C_4 = \ldots$

$g = a\ b$
$p = a + b$

16 Bit Carry Look Ahead

- $g_i = a_i\, b_i$
- $p_i = a_i + b_i$
- $c_1 = g_0 + p_0\, c_0$
- $c_2 = g_1 + p_1\, g_0 + p_1 p_0 c_0$
- $c_3 = g_2 + p_2\, g_1 + p_2\, p_1\, g_0 + p_2\, p_1 p_0\, c_0$

1 3

- $c_4 = \underbrace{g_3 + p_3 g_2 + p_3 p_2\, g_1 + p_3\, p_2\, p_1\, g_0}_{G_0} + \underbrace{p_3\, p_2\, p_1 p_0}_{P_0}\, \boxed{c_0}$

3 1 1 1 2 1

- $G_0 = g_3 + p_3\, g_2 + p_3\, p_2\, g_1 + p_3\, p_2\, p_1\, g_0$ $P_0 = p_3\, p_2\, p_1\, p_0$ 1 1
- $G_1 = g_7 + p_7\, g_6 + p_7\, p_6\, g_5 + p_7\, p_6\, p_5\, g_4$ $P_1 = p_7\, p_6\, p_5\, p_4$
- $G_2 = g_{11} + p_{11}\, g_{10} + p_{11}\, p_{10}\, g_9 + p_{11}\, p_{10}\, p_9\, g_8$ $P_2 = p_{11}\, p_{10}\, p_9\, p_8$
- $G_3 = g_{15} + p_{15}\, g_{14} + p_{15}\, p_{14}\, g_{13} + p_{15}\, p_{14}\, p_{13}\, g_{12}$ $P_3 = p_{15}\, p_{14}\, p_{13}\, p_{12}$

16 Bit Carry Look Ahead (Cont.)

$C_4 \leftarrow \boxed{3}$

C_{16} $C'_4 \leftarrow \boxed{5}$

$C_{64} \leftarrow C'_4 = \boxed{7}$

- $C_1 = G_0 + P_0C_0$
- $C_2 = G_1 + P_1C_1$ $C_2 =$
- $C_3 = G_2 + P_2C_2$ $C_3 =$
- $C_4 = G_3 + P_3C_3$ $C_4 =$

- $C_1 = c_4 = G_0 + P_0\ c_0$
- $C_2 = c_8 = G_1 + P_1\ G_0 + P_1\ P_0\ c_0$
- $C_3 = c_{12} = G_2 + P_2\ G_1 + P_2\ P_1\ G_0 + P_2\ P_1\ P_0\ c_0$
- $C_4 = c_{16=}\ \overbrace{G_3 + P_3G_2 + P_3\ P_2\ G_1 + P_3\ P_2\ P_1\ G_0}^{G'_0} + \overbrace{P_3\ P_2\ P_1\ P_0}^{P'_0}\ c_0$

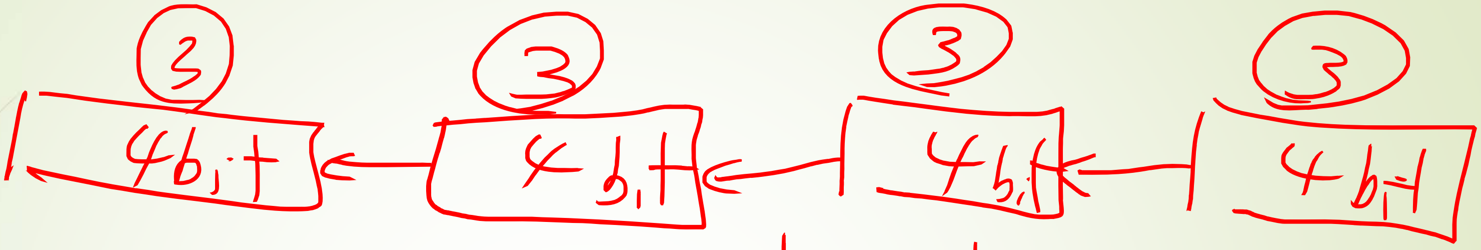
4 $\boxed{3}$ 2

$\boxed{1}$

$\boxed{1}$

$\boxed{5}$

16 Bit Carry Look Ahead (Cont.)



12 gate delays

32 Bit

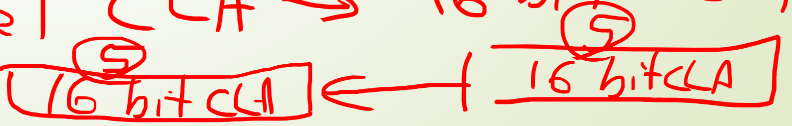
* 8X3=24 gate delay

if we use 4 bitCLA

* ripple 32*2=64

* 2 level CLA → 16 bit CLA 5

10 gate delay

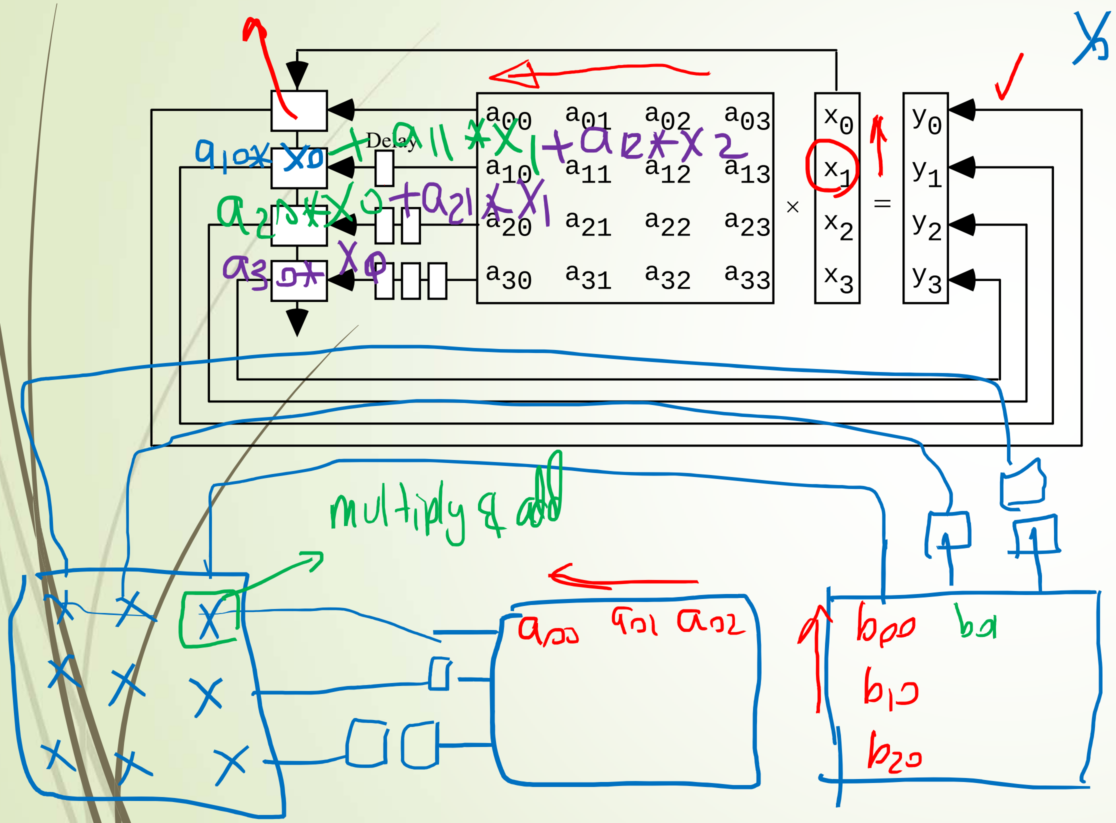


SIMD multiply & add

6) The following diagram depicts a vector processor for matrix – vector multiplier. Devise a vector processor for a matrix – matrix multiplication.

$a_{00} * X_0 + a_{01} * X_1 + a_{02} * X_2 + a_{03} * X_3$

$Y_0 = a_{00} * X_0 + a_{01} * X_1 + a_{02} * X_2 + a_{03} * X_3$

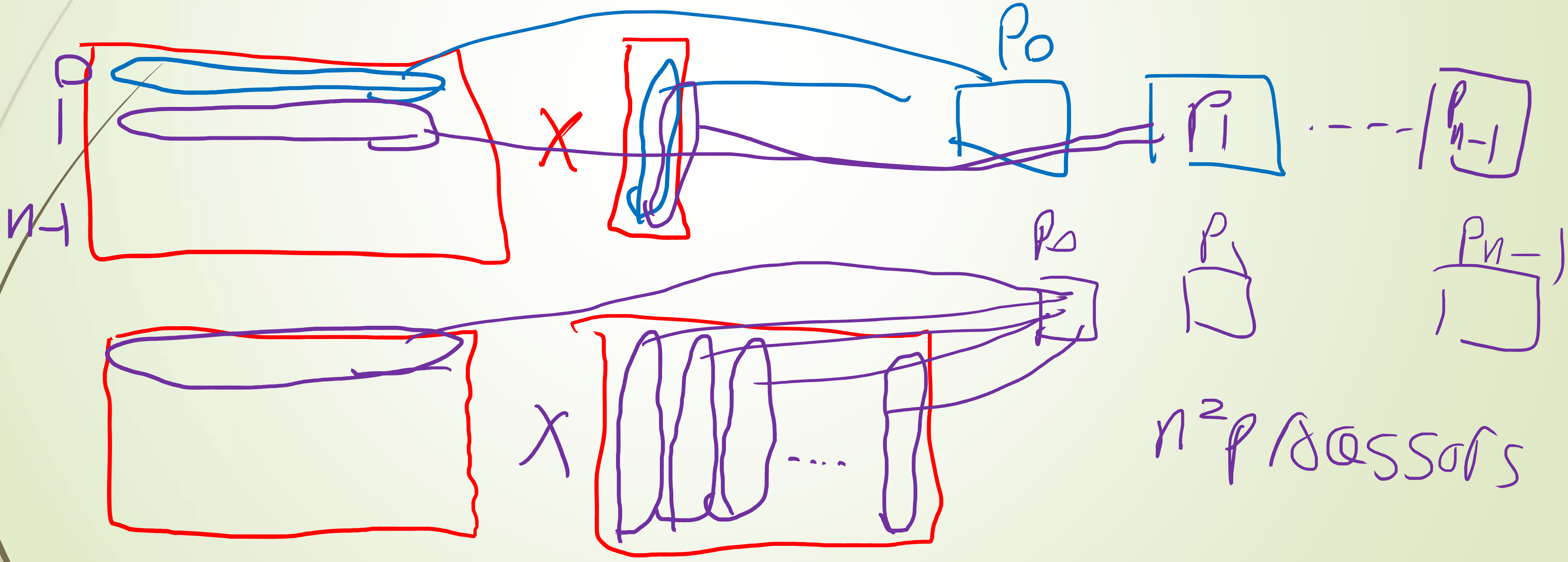


$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{bmatrix}$$

SIMD SPMD

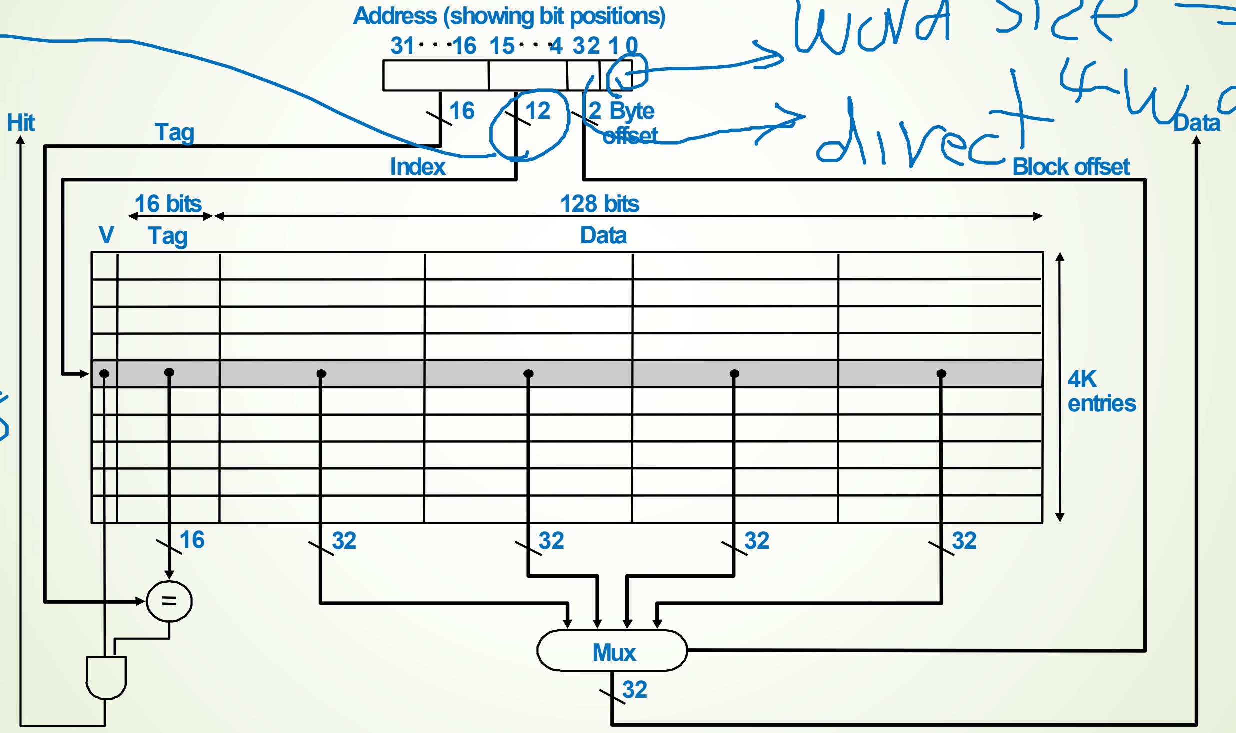
single instruction single program

on different data



Direct Mapped Cache

- Taking advantage of spatial locality:



word size → 4 Bytes

direct 4-way

$2^{12} = 4K$ rows

4K × 4 × 4
way Bytes

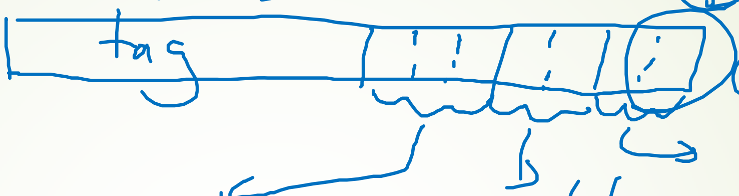
64K cache

4 way direct cache word size 4 bytes

8 raws

total cach

32-7=25



Index

128	64	32	16	8	4	2	1
~~256~~ A_6	A_5	A_4	A_3	A_2	A_1	A_0	

M

24 0 0 1 x x —

H

28 001

select word 12 000 11

M

4 H

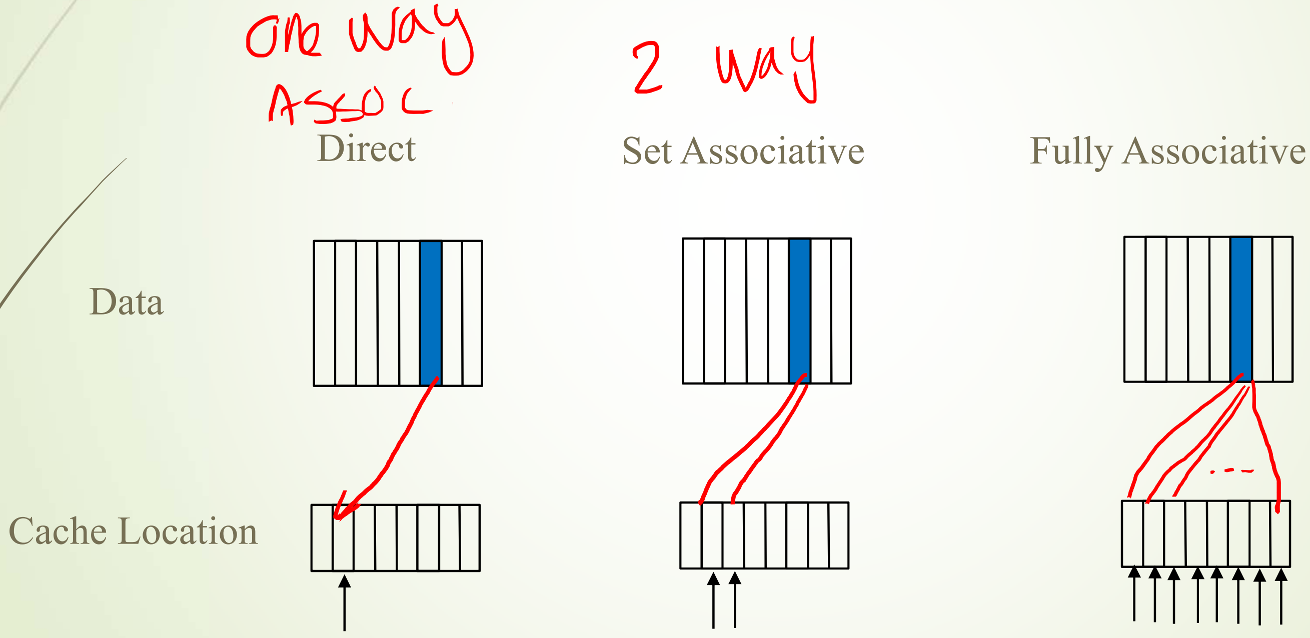
8 H

132 M 000 01 —

	11	10	01	00
000	~~12~~ 140	~~8~~ 136	~~4~~ 132	~~0~~ 128
001	28	24	20	16
010				
011				
100				
101				
110				
111				

Decreasing Miss Ratio with Associativity

Associativity: Reducing cache misses by more flexible placement of blocks



One way ASSOC

2 way

Direct

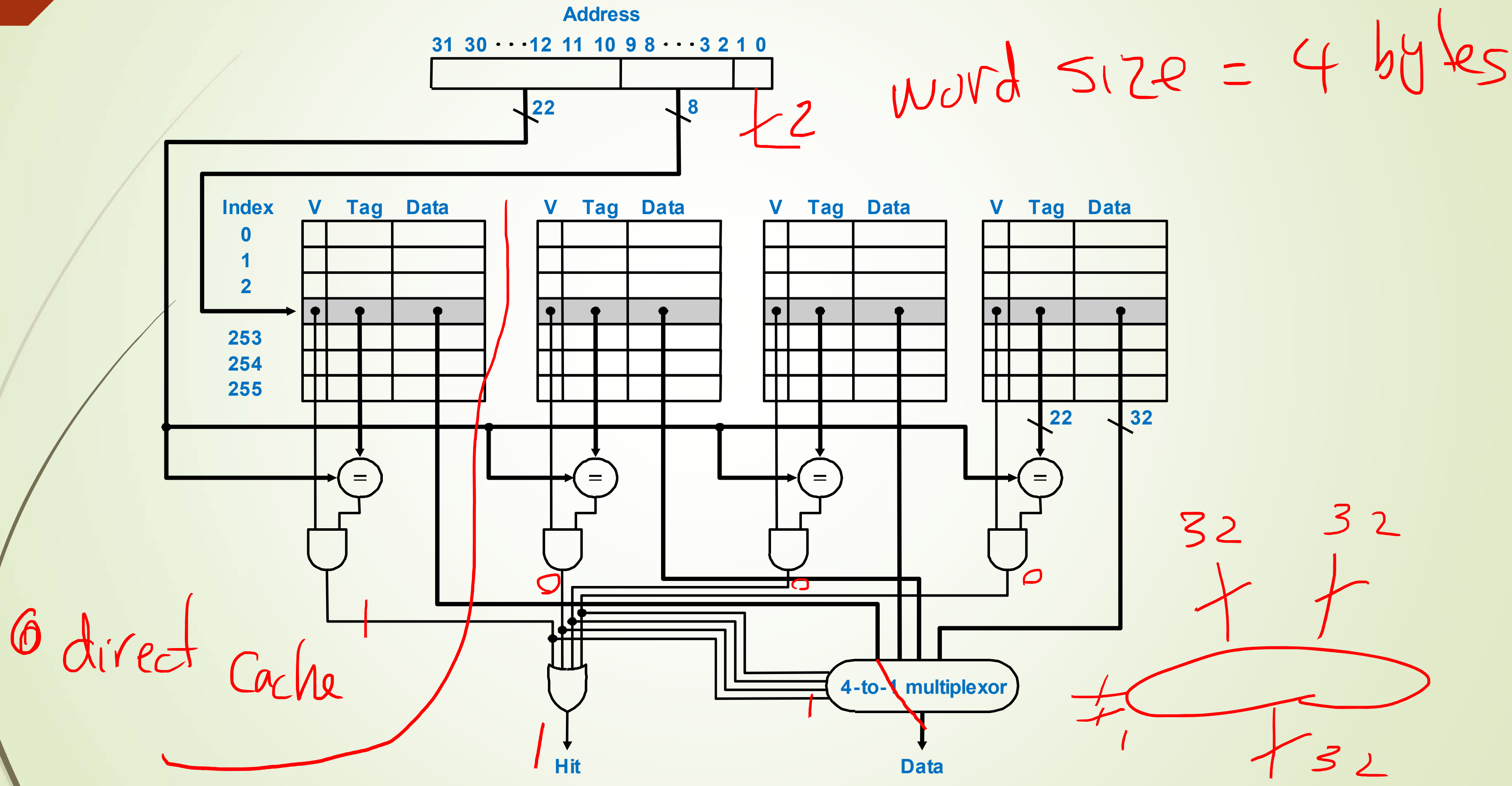
Set Associative

Fully Associative

Data

Cache Location

4-Way Associative Cache Organization



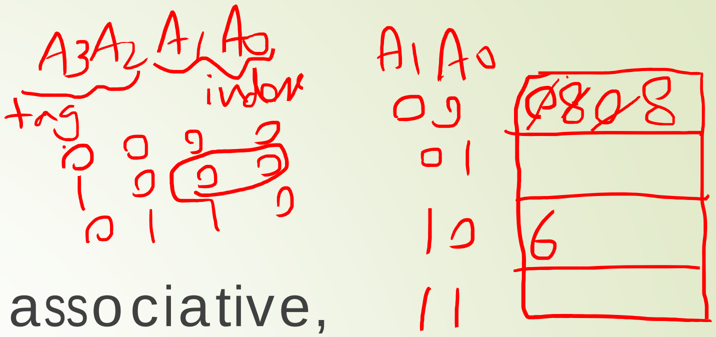
word size = 4 bytes

6 direct cache

Associative Caches

- Fully associative
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
  - Comparator per entry (expensive)
- *n*-way set associative
  - Each set contains *n* entries
  - Block number determines which set
    - (Block number) modulo (#Sets in cache)
  - Search all entries in a given set at once
  - *n* comparators (less expensive)

Associativity Example



- Assume word size = 1 byte
- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8
- Direct mapped

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	

Associativity Example

LIFO

A3	A2	A1	A0
0	0	0	0
1	0	0	0
0	1	1	0

A0		
0	0	~~8~~~~6~~8
1		

- 2-way set associative

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	**Mem[0]**			
8	0	miss	Mem[0]	**Mem[8]**		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	Mem[0]	**Mem[6]**		
8	0	miss	**Mem[8]**	Mem[6]		

- Fully associative

0	8	6	

Block address		Hit/miss	Cache content after access			
0		miss	**Mem[0]**			
8		miss	Mem[0]	**Mem[8]**		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	**Mem[6]**	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

Spectrum of Associativity

- Assume one byte word size. $A_2 A_1 A_0$
- For a cache with 8 entries. Assume using the "least recently used" replacement strategy 7, 4, 17, 12, 4, 25, 7, 13

M M M

H

One-way set associative (direct mapped)

Block	Tag	Data
0		
1	~~2~~	~~17~~ 25
2		
3		
4		~~4~~ ~~12~~ 4
5		13
6		
7		7

Two-way set associative

AA_0

Set	Tag	Data	Tag	Data
0		4		12
1	~~4~~	~~17~~ 13		25
2				
3		7		

Four-way set associative

A_0

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0		4		12				
1	8	7		17		25		13

Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data
	7		4	17	17		12		25		13				

7
4
17
12

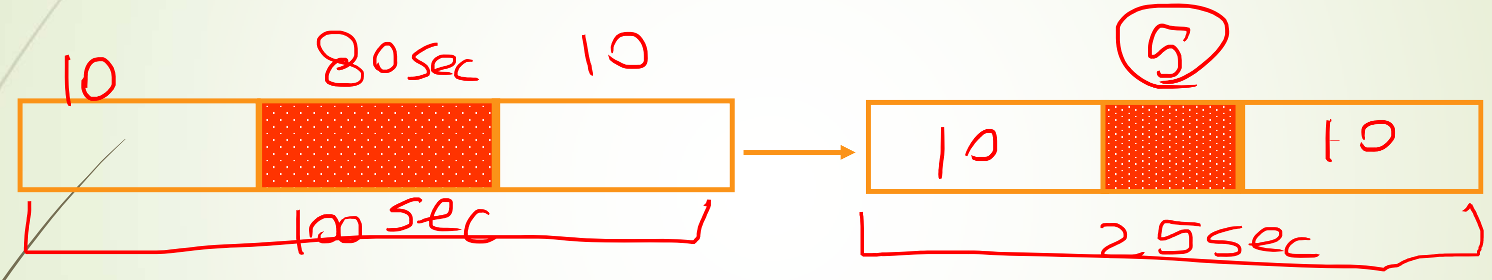
32 16 8 4 2 1
A_5 A_4 A_3 A_2 A A_0
1 1 1
1 0 0
1 0 0 0 1
0 1 1 0 0
1 1 0 0 1
1 1 0 1

Pitfall: Amdahl's Law

$$T_{improved} = \frac{T_{affected}}{\text{improvement factor}} + T_{unaffected}$$

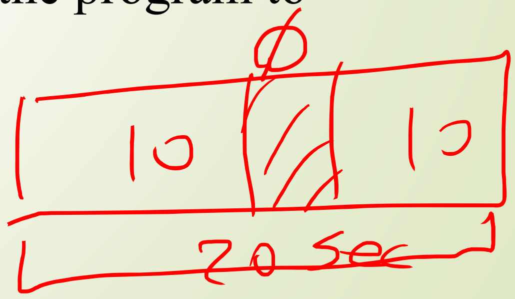
$\frac{80}{5} = 16$

X



Example: Suppose a program runs in 100 seconds on a machine, with multiply responsible for 80 seconds of this time. How much do we have to improve the speed of multiplication if we want the program to run 4 times faster?

5X faster

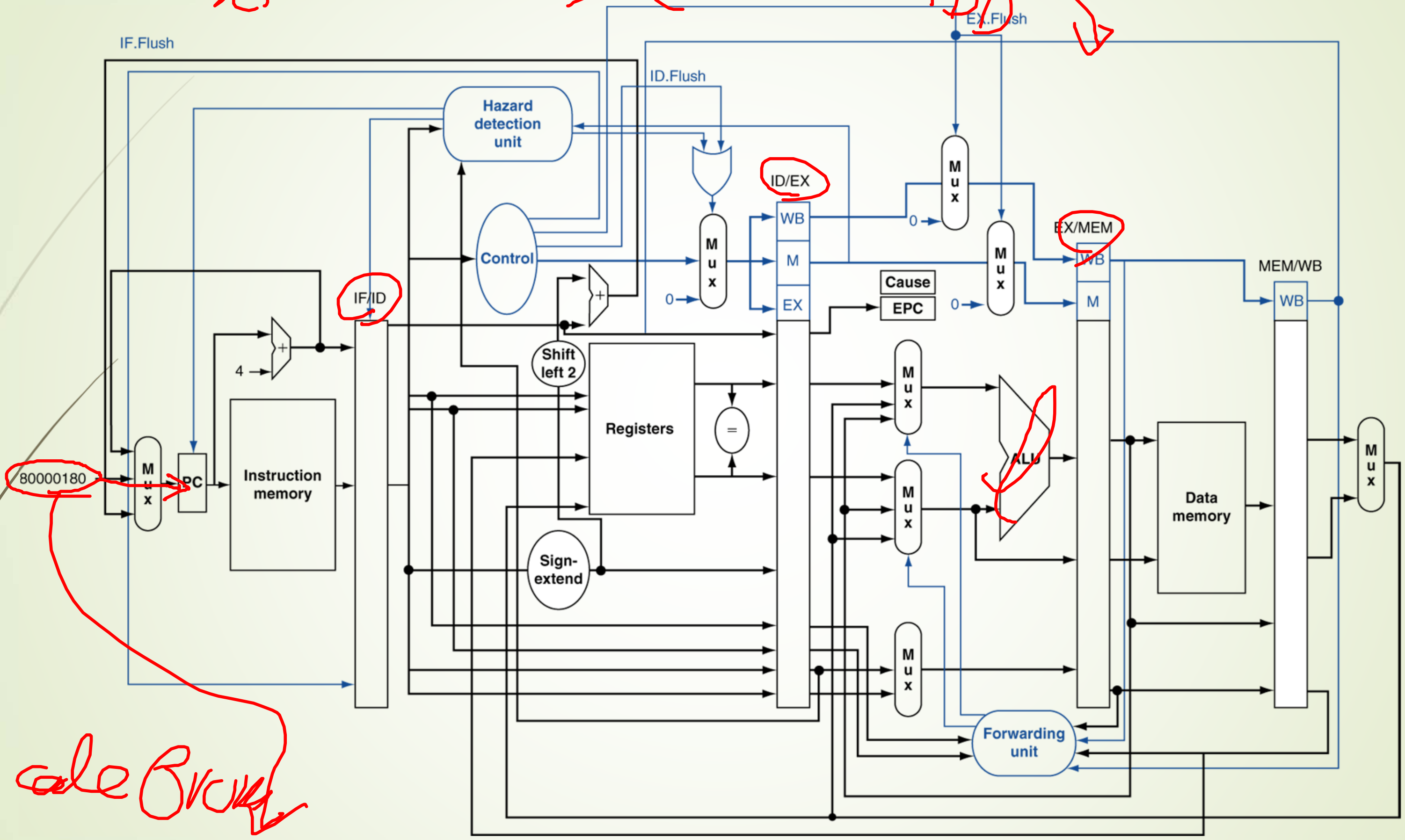


Pipeline with Exceptions

nop ~~next~~

nop ~~no~~

nop ~~ADD~~



op code Branch

Pitfall: Amdahl's Law

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$$

$$\frac{100}{4} = \frac{80}{n} + 20$$

- n = 16

- How much improvement in multiply performance to get 5× overall?

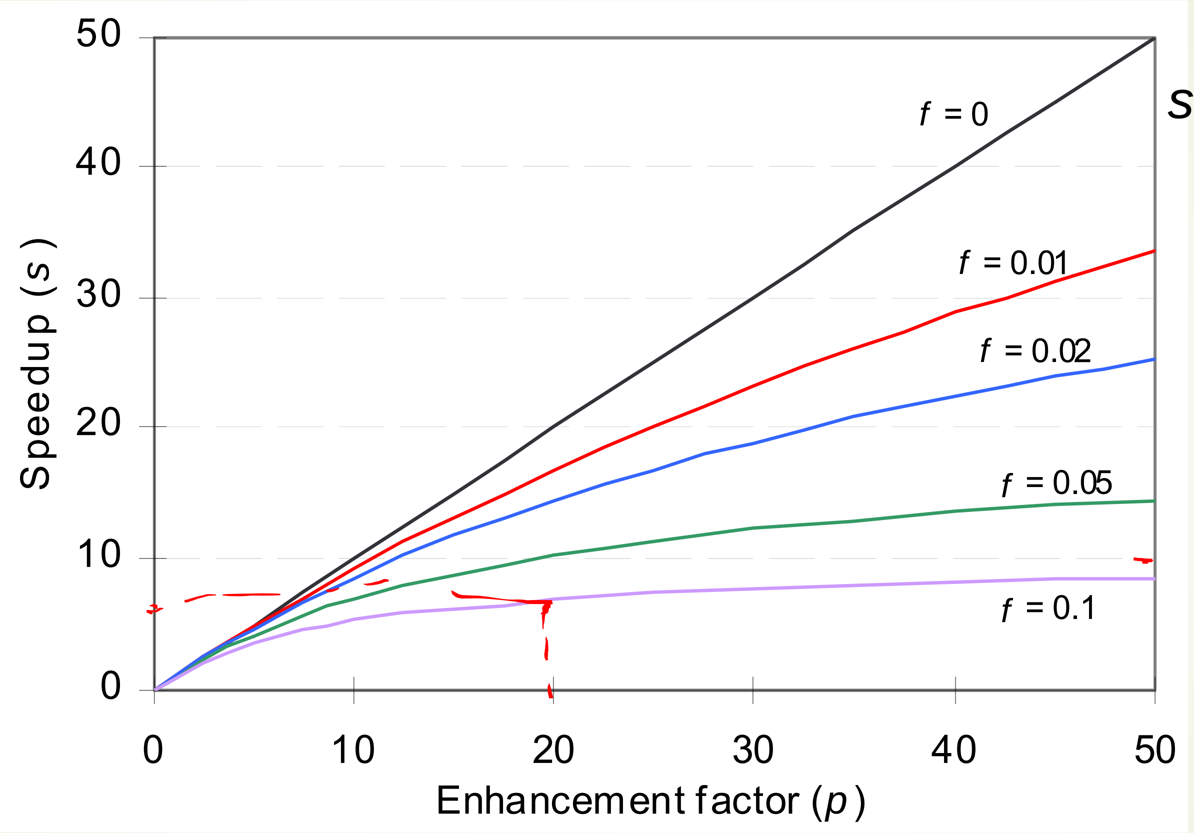
$$20 = \frac{80}{n} + 20$$

- Can't be done!

- Improving an aspect of a computer and expecting a proportional improvement in overall performance

- Corollary: make the common case fast

Amdahl's Law – Parallel Processing Version



$$s = \frac{\text{Exec Time 1 Processor}}{\textit{Exec Time P Processors}}$$

f = fraction sequential

p = speedup of the rest

$$s = \frac{1}{f + (1 - f)/p}$$

$\leq min(p, 1/f)$

10%

$f = .1$

$$s = \frac{1}{.1 + \frac{.9}{p}}$$

$$s = \frac{1}{.1 + \frac{.9}{20}}$$

Limit on speed-up according to Amdahl's law.