CS/COE1541: Introduction to Computer Architecture

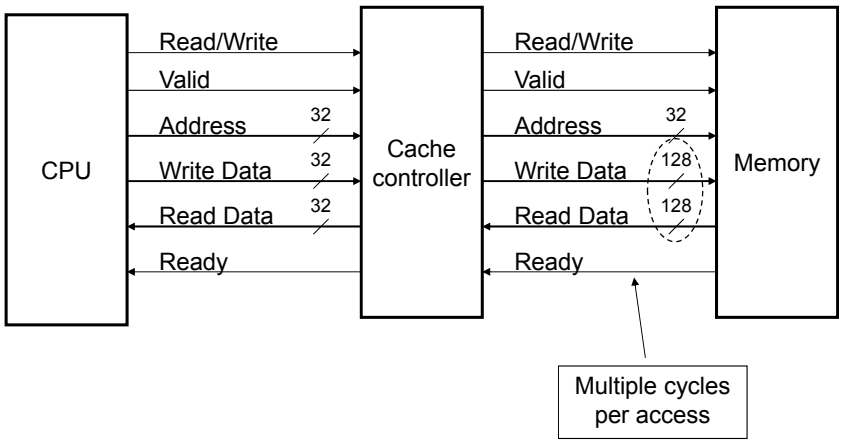
Dept. of Computer Science
University of Pittsburgh

http://www.cs.pitt.edu/~melhem/courses/1541p/index.html

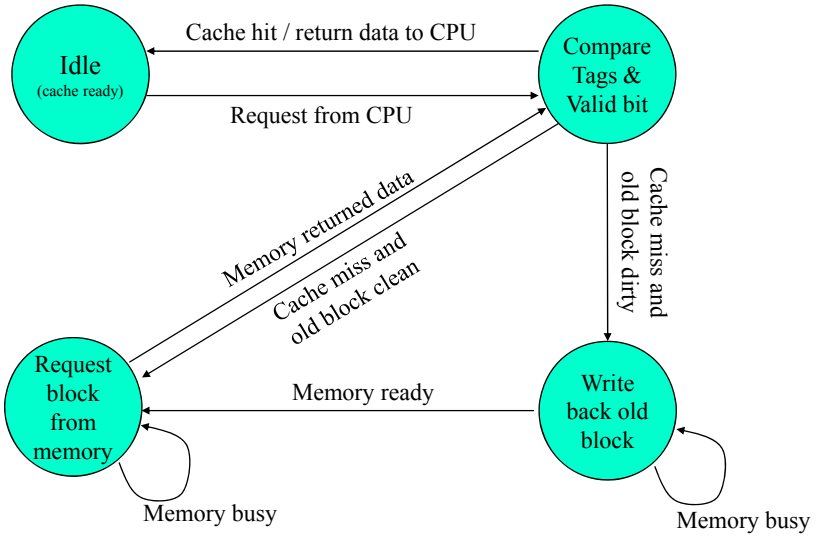
Chapter 5: Exploiting the Memory Hierarchy
Lecture 4

Lecturer: Rami Melhem

CPU-cache-memory Interface Signals



Cache Controller FSM (sec. 5.9)



Software optimization 1: loop interchange (sec. 5.4)

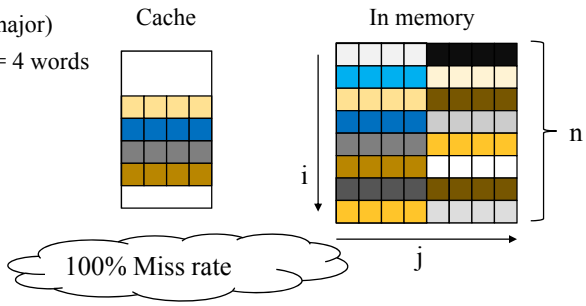
Matrix A is stored Row-wise (row major)

Fully associative cache, block size = 4 words

Cache size < 4n words

```
for (j = 0; j< n; j = j+1)
  for (i = 0; i < n; i = i+1)
      C =+ A[i][j];
```

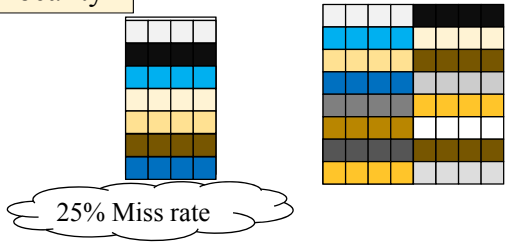
Column-wise memory access



Take advantage of spatial locality

Row-wise memory access

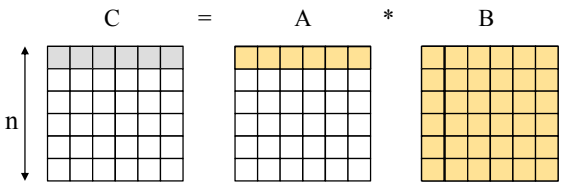
```
for (i = 0; i < n; i = i+1)
  for (j = 0; j < n; j = j+1)
      C = + A[i][j];
```



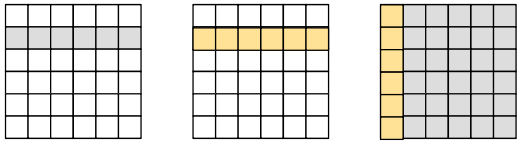
Software optimization 2: blocking (partitioning)

Matrix multiplication

```
for (i = 0 ; i < n ; i++)
  for (j = 0 ; j < n ; j ++)
    {r = 0;
     for (k = 0; k < n; k++)
        r = r + A[i][k]*B[k][j];
     C[i][j] =  r;  };
```



Data used when i = 0, j = 0, … , n-1



Data used when i = 1, j = 0, … , n-1

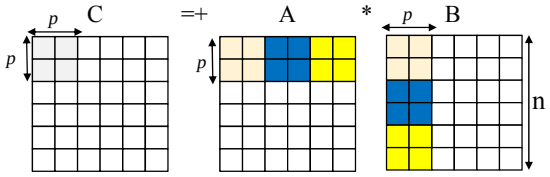
Assume:
A fully associative cache
Block size = 1 word
Cache size < n^2

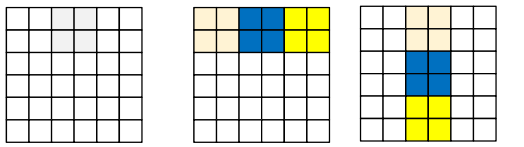
- One row of A will fit in the cache and be repeatedly used (perfect reuse)
- B will not fit in cache and hence a column of B will be evicted before reuse
- Every element of B will be used only once when brought to the cache

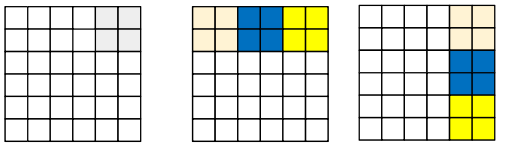
Software optimization through blocking (partitioning)

Partition the matrices into submatrices of size p x p

```
for (si = 0; si < n; si =+ p)
 for (sj = 0; sj < n; sj =+ p)
  for (sk = 0; sk < n; sk =+ p)
      for (i=si ; i < si+p ; i++)
       for (j=sj ; j < sj+p ; j++)
          {r = 0;
           for (k=sk; k < sk+p; k++)
                r = r + A[i][k]*B[k][j];
           C[i][j] = C[i][j] + r;
          };
```





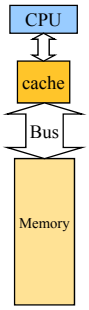


If cache size > $p * n + p^2$, then

- A will be perfectly reused
- Each element of B will be reused "p" times (reduce miss rate)

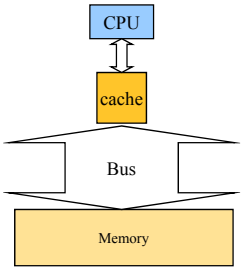
HW optimization: Interleaving for faster block access

- Accessing **a block of K words** should be faster than accessing **K words** at different times. Otherwise, fetching an entire block does not pay off – actually may hurt.
- **EX:** 1 cycle to send a word or an address and 15 cycles to fetch a word in memory (only 7 cycles if "open row" policy and the row is already open in the row buffer)



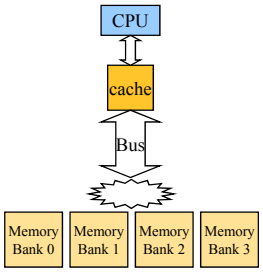
one-word wide memory

Time to access 4 words	Time to access a block (4 words)
4*(1+15+1)=68 cycles	(1+15+1)+3(1+7+1)=44 cycles
Rows will not be open if accesses are not strictly consecutive	The row will be open when accessing the last three words



4-word wide organization
(wide bus and memory ports)

Time to access a block (4 words)
= 1 + 15 + 1 = 17 cycles

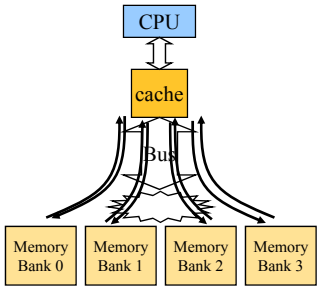


Interleaved memory

Time to access a block (4 words)
= (1 + 15 + 1) + 3 = 20 cycles

Interleaved memory

- T = 1 send request to bank 0
- T = 2 send request to bank 1
- T = 3 send request to bank 2
- T = 4 send request to bank 3
- T = 16 data ready at Bank 0 and received at T = 17
- T = 17 data ready at Bank 0 and received at T = 18
- T = 18 data ready at Bank 0 and received at T = 19
- T = 19 data ready at Bank 0 and received at T = 20



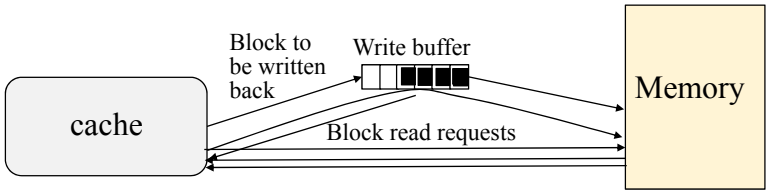
Interleaved memory

Time to access memory =
(1 + 15 + 1) + 3 = 20 cycles

HW optimization: using write buffers to reduce miss penalty

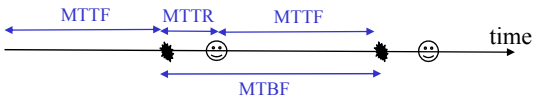
- Writing back an evicted block before reading a block increases the miss penalty
- Can read the requested block before writing back the old if a write buffer is used
- Priority is given to reading blocks in order to reduce miss penalty
- Blocks in the buffer are written back whenever there are no read requests
- Consecutive read requests will results in pending write backs in the write buffer

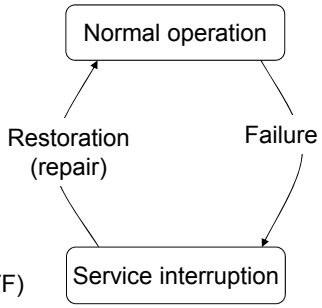
- For correctness, before sending any read request to memory, we have to check the write buffer
- If block is still in the write buffer, do not send the request to memory.



Dependable memory hierarchy (sec. 5.5)

- Fault: failure of a component
- Error: manifestation of a fault
- Faults may or may not lead to system failure



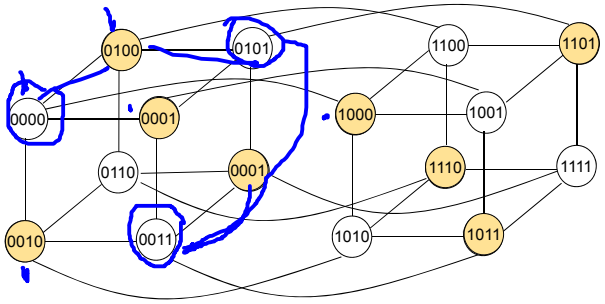


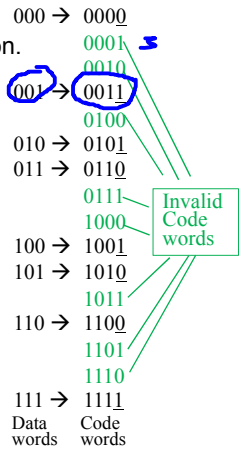
- **Reliability measure**: mean time to failure (MTTF)
- **Repair efficiency**: mean time to repair (MTTR)
- Mean time between failures

  MTBF = MTTF + MTTR
- Availability = MTTF / MTBF
- Improving Availability
  - Increase MTTF: fault avoidance, fault tolerance, fault forecasting
  - Reduce MTTR: improved tools and processes for diagnosis and repair

Error detection Codes (even parity codes)

- Hamming distance: Number of bits that are different between two bit patterns
  Example: distance between 1001 and 1010 is 2
- Codes with min. distance = 2 provides single bit error detection.
  Example: even parity code

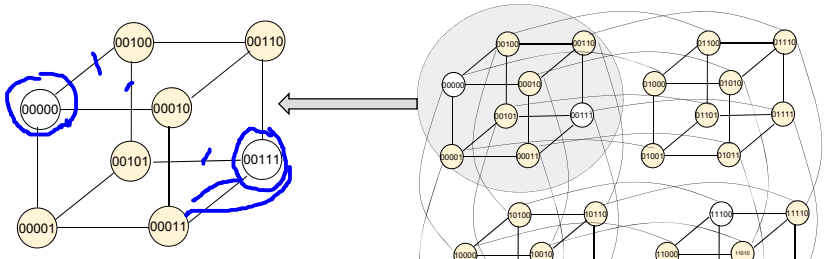




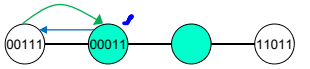
- Eight of the sixteen 4-bit code words are invalid
- Any single bit flip in a valid code will produce an invalid code
  - Hence, single error detection --- but cannot correct the error
  - Note that two errors will go undetected

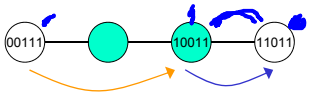
Error correcting Codes

Minimum distance = 3 provides single error correction (SEC)



Single fault (can be corrected)





Double faults (corrected to wrong code)

Data words	Code words
00	00000
01	00111
10	11100
11	11011

2-bit → 5-bit encoding

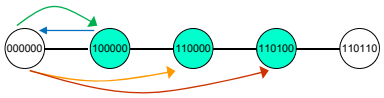
Error detection and correction

Minimum distance = 4 provides single error correction (SEC) and double error detection (DEC)

00 → 000000
01 → 001111
10 → 111001
11 → 110110

2-bit → 6-bit encoding

Single fault (can be corrected)



Double faults (will be detected)

Three faults (corrected to wrong code)

Hamming (Single error correcting) code

- To calculate code a 12-bits code word from an 8-bits data word:
  - Number the bits of the code words from 1 to 12
  - All bit positions that are a power of 2 are parity bits, the others are data bits
  - Each parity bit is set so that a certain group of data bits have even parity.

Bit position:		1	2	3	4	5	6	7	8	9	10	11	12
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverage	p1	x		x		x		x		x		x	
	p2		x	x			x	x			x	x	
	p4				x	x	x	x					x
	p8								x	x	x	x	x

Encoded data bits		p1	p2	1	p4	0	1	0	p8	0	0	1	1
Parity bit coverage	p1	0		1		0		0		0		1	
	p2		1	1			1	0			0	1	
	p4				0	0	1	0					1
	p8								0	0	0	1	1

Example:
Data bits 10100011 are encoded as 011001000011

Note: the minimum distance of the Hamming code = 3

Hamming (Single error correcting) code

- If an error occurs in any of the 12 bits

  EXAMPLE: 011001000011 becomes 011001000111

- Check the parity groups and compute the syndrome bits, s_1, s_2, s_3, s_4

	1	2	3	4	5	6	7	8	9	10	11	12
	0	1	1	0	0	1	0	0	0	1	1	1
s_1	0		1		0		0		0		1	
s_2		1	1			1	0			1	1	
s_3				0	0	1	0					1
s_4								0	0	1	1	1

$s_1 = 0$ (parity is correct)
$s_2 = 1$ (parity is not correct)
$s_3 = 0$ (parity is correct)
$s_4 = 1$ (parity is not correct)

- If all syndrome bits are zeroes, then there is no error
- Otherwise, the syndrome bits indicate the position of the bit in error
- In our example s_4, s_3, s_2, s_1 = 1010 = ten → bit ten is the wrong bit

Not magic!!
There is a theory behind that

Hamming SEC/DED Code

- Hamming code cannot detect two errors (distance < 4)
- Add an additional parity bit for the whole word (p_n)
- Make Hamming distance = 4
- Decoding:
  - No error in p_n and syndrome = 0 → no error
  - Error in p_n and syndrome > 0 → single correctable error
  - No error in p_n and syndrome > 0 → double errors (uncorrectable)
  - Error in p_n and syndrome = 0 → error in the SEC parity bit
- Note: ECC DRAM uses SEC/DED with 8 bits (7 for syndrome and 1 for p_n) protecting each 64 bits