CPU-cache-memory Interface Signals

CPU
- Read/Write
- Valid
- Address 32
- Write Data 32
- Read Data 32
- Ready

Cache controller
- Read/Write
- Valid
- Address 32
- Write Data 32
- Read Data 32
- Ready

Memory
- Read/Write
- Valid
- Address 32
- Write Data 128
- Read Data 128
- Ready

Multiple cycles per access
Cache Controller FSM (sec. 5.9)

 Idle (cache ready)

 Compare Tags & Valid bit

 Request block from memory

 Write back old block

 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];

 100% Miss rate

 25% Miss rate
Software optimization 2: blocking (partitioning)

Matrix multiplication

\[
C[i][j] = \sum_{k=0}^{n-1} A[i][k] \times B[k][j]
\]

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] \times B[k][j];
 C[i][j] = r;  };

Data used when \( i = 0, j = 0, \ldots, n-1 \)

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 \( p \times p \)

\[
C[i][j] = \sum_{k=0}^{n-1} A[i][k] \times B[k][j]
\]

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] \times B[k][j];
 C[i][j] = C[i][j] + r;  };

If cache size > \( p \times n + p^2 \), 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)

<table>
<thead>
<tr>
<th>One-word wide memory</th>
<th>4-word wide organization (wide bus and memory ports)</th>
<th>Interleaved memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to access 4 words</td>
<td>Time to access a block (4 words)</td>
<td>Time to access a block (4 words)</td>
</tr>
<tr>
<td>4*(1+15+1)=68 cycles</td>
<td>(1+15+1)+3(1+7+1)=44 cycles</td>
<td>(1+15+1)+3=20 cycles</td>
</tr>
<tr>
<td>Rows will not be open if accesses are not strictly consecutive</td>
<td>The row will be open when accessing the last three words</td>
<td></td>
</tr>
</tbody>
</table>

**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

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 result 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

\[ \text{MTBF} = \text{MTTF} + \text{MTTR} \]

- 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)
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

Single fault (can be corrected)
Double faults (will be detected)
Three faults (corrected to wrong code)

2-bit → 6-bit encoding

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.

<table>
<thead>
<tr>
<th>Bit position:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoded data bits</td>
<td>p1</td>
<td>p2</td>
<td>d1</td>
<td>p4</td>
<td>d2</td>
<td>d3</td>
<td>d4</td>
<td>p8</td>
<td>d5</td>
<td>d6</td>
<td>d7</td>
<td>d8</td>
</tr>
<tr>
<td>Parity bit coverage</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>p1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>p2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>p2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>p4</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>p4</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>p8</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>p8</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Example:
Data bits 10100011 are encoded as 01100100011

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, s₁, s₂, s₃, s₄

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₁</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>s₂</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>s₃</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>s₄</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

  - s₁ = 0 (parity is correct)
  - s₂ = 1 (parity is not correct)
  - s₃ = 0 (parity is correct)
  - s₄ = 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 s₄, s₃, s₂, s₁ = 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 (pₙ)
- Make Hamming distance = 4
- Decoding:
  - No error in pₙ and syndrome = 0 → no error
  - Error in pₙ and syndrome > 0 → single correctable error
  - No error in pₙ and syndrome > 0 → double errors (uncorrectable)
  - Error in pₙ and syndrome = 0 → error in the SEC parity bit

- Note: ECC DRAM uses SEC/DED with 8 bits (7 for syndrome and 1 for pₙ) protecting each 64 bits