Caching

Definitions:

- **Collision** When two blocks of memory are mapped to the same location in the cache.
- **Block** A block is a group of words that are stored together in memory and in the cache.
- **Set** A set is a group of blocks that are stored together in the cache.
- **Index** The index is the part of the address that determines which set the block is stored in.
- **Tag** The tag is the part of the address that determines which block in the set the word is stored in.

Direct Mapped Cache

Summary: Direct Map Caching is a means of caching in which each block of memory is mapped to a specific location in the cache. This means that each block of memory can only be stored in one location in the cache. This is the simplest form of caching, but it is also the least efficient.

Example:

Given:

- Cache Details:
  - 8 blocks
  - 4 words per block
  - 32 bits per word (This is the default size we have been using for all of our examples)
- Word Addresses:
  - 43
  - 171
  - 45
  - 178
  - 179
  - 5
  - 174
  - 46
  - 4
  - 181
  - 65
  - 180

1. **Convert to Binary Addresses:** The first thing we need to do is convert the word addresses into binary addresses. These addresses are going to be 30 bits long since each word is 32 bits long (and we need to subtract 2 bits for the word offset).

Word Address	Binary Address
43	.... 0010 1011
171	.... 1010 1011
45	.... 0010 1101
178	.... 1011 0010
179	.... 1011 0011
5	.... 0000 0101
174	.... 1010 1110
46	.... 0010 1110
4	.... 0000 0100
181	.... 1011 0101
65	.... 0100 0001
180	.... 1011 0100

2. **Decompose the Binary Addresses:** In order to cache the addresses, we need to decompose them into their word number, index, and tag (from right to left).
   - **Word Number** $2^x =$ (Words per Block)
     - Since there are 4 words per block, we can determine the number of bits needed to represent the word number by taking the $2^x = 4$ and solving for x. This gives us $x = 2$.

- **Index** 2^x = (Blocks in Cache)
  - Since there are 8 blocks in the cache, we can determine the number of bits needed to represent the index by taking $2^x = 8$ and solving for x. This gives us $x = 3$.
- **Tag** (Address Length) − (Word Number Length) − (Index Length)
  - Since the address length is 30 bits, the word number length is 2 bits, and the index length is 3 bits, we can determine the number of bits needed to represent the tag by taking $30 - 2 - 3 = 25$.

Number	Address	Tag	Index	Word
43	... 0000 0010 1011	... 0001	010	11
171	... 0000 1010 1011	... 0101	010	11
45	... 0000 0010 1101	... 0001	011	01
178	... 0000 1011 0010	... 0101	001	10
179	... 0000 1011 0011	... 0101	001	11
5	... 0000 0000 0101	... 0000	001	01
174	... 0000 1010 1110	... 0101	110	10
46	... 0000 0010 1110	... 0001	011	10
4	... 0000 0000 0100	... 0000	000	00
181	... 0000 1011 0101	... 0101	101	01
65	... 0000 0100 0001	... 0010	000	01
180	... 0000 1011 0100	... 0101	101	00

3. **Create the Cache:** Cache Initial State:

Index	Tag	Word 3	Word 2	Word 1	Word 0
000	...	...	...	...	...
001	...	...	...	...	...
010	...	...	...	...	...
011	...	...	...	...	...
100	...	...	...	...	...
101	...	...	...	...	...
110	...	...	...	...	...
111	...	...	...	...	...

4. **Map the Addresses to the Cache:** For each address, we need to determine if it is a hit or a miss. We do this by checking the tag of the entry in the cache that corresponds to the index of the address. If the tag matches the tag of the address, then it is a hit. If the tag does not match the tag of the address, then it is a miss. If the entry in the cache is empty, then it is a miss.
5. **Update the Cache:** If the address is a miss, then we need to update the cache. We do this by replacing the word in the cache that corresponds to the index, tag, and word number of the address with the new entry. Then we need to update the rest of the words in the block with the surrounding words in memory.

Final Cache State:

Index	Tag	Word 3	Word 2	Word 1	Word 0
000	0010	67	66	65	64
001	0000	7	6	5	4
010	0101	171	170	169	168
011	0001	47	46	45	44
100	0101	179	178	177	176
101	0101	183	182	181	180
110	None	.....	.....	.....	.....
111	None	.....	.....	.....	.....

This cache resulted in a 3/12 hit rate.

Tricky Things to Remember:

- Check whether we are using byte addresses or word addresses

Two-Way Set Associative Cache

Summary: A two-way set associative cache is a cache in which each index cooresponds to a set composed of two blocks. This means that there are two possible places for a block to be stored in the cache.

Given:

- Cache Details:
  - 8 blocks
  - 4 words per block
  - 32 bits per word (This is the default size we have been using for all of our examples)
- Word Addresses:
  - 43
  - 171
  - 45
  - 178
  - 179
  - 5
  - 174
  - 46
  - 4
  - 181
  - 65
  - 180

1. **Convert to Binary Addresses:** The first thing we need to do is convert the word addresses into binary addresses. These addresses are going to be 30 bits long since each word is 32 bits long (and we need to subtract 2 bits for the word offset).

Word Address	Binary Address
43	.... 0010 1011
171	.... 1010 1011
45	.... 0010 1101
178	.... 1011 0010
179	.... 1011 0011
5	.... 0000 0101
174	.... 1010 1110
46	.... 0010 1110
4	.... 0000 0100
181	.... 1011 0101
65	.... 0100 0001
180	.... 1011 0100

   - **Note:** We are using word addresses, if we are using byte addresses, then we would need to multiply the word address by 4 (add 2 zeros to the end of the binary address).

2. **Decompose the Addresses:** In order to cache the addresses, we need to decompose them into their word number, index, and tag (from right to left).
   - **Word Number** $2^x =$ (Words per Block)
     - Since there are 4 words per block, we can determine the number of bits needed to represent the word number by taking the $2^x = 4$ and solving for x. This gives us $x = 2$.
   - **Index** $2^x =$ (Blocks in Cache)
     - Since there are 8 blocks in the cache (Divided into sets of 2), we can determine the number of bits needed to represent the index by taking $2^x = \frac{8}{2}$ and solving for x. This gives us $x = 2$.
   - **Tag** (Address Length) − (Word Number Length) − (Index Length)
     - Since the address length is 30 bits, the word number length is 2 bits, and the index length is 2 bits, we can determine the number of bits needed to represent the tag by taking $30 - 2 - 2 = 26$.

Number	Address	Tag	Index	Word
43	... 0000 0010 1011	 0010	10	11
171	... 0000 1010 1011	 1010	10	11
45	... 0000 0010 1101	 0010	11	01

Number	Address	Tag	Index	Word
178	... 0000 1011 0010	 1010	01	10
179	... 0000 1011 0011	 1010	01	11
5	... 0000 0000 0101	 0000	01	01
174	... 0000 1010 1110	 1011	10	10
46	... 0000 0010 1110	 0010	11	10
4	... 0000 0000 0100	 0000	00	00
181	... 0000 1011 0101	 1011	01	01
65	... 0000 0100 0001	 0100	00	01
180	... 0000 1011 0100	 1011	01	00

3. **Create the Cache:** Cache Initial State:

Index	Tag	Word 3	Word 2	Word 1	Word 0
00	...	...	...	...	...
00	...	...	...	...	...
01	...	...	...	...	...
01	...	...	...	...	...
10	...	...	...	...	...
10	...	...	...	...	...
11	...	...	...	...	...
11	...	...	...	...	...

4. **Map the Addresses to the Cache:** For each address, we need to determine if it is a hit or a miss. We do this by checking both tags in the set that corresponds to the index of the address. If either of the tags matches the tag of the address, then it is a hit. If neither of the tags matches the tag of the address, then it is a miss. If both of the entries in the set are empty, then it is a miss.
5. **Update the Cache:** If the address is a miss, then we need to update the cache. We do this by replacing the word in the cache that corresponds to the index, tag, and word number of the address with the new entry. Then we need to update the rest of the words in the block with the surrounding words in memory.

   Final Cache State:

Index	Tag	Word 3	Word 2	Word 1	Word 0
00	...	...	...	...	...
00	 0100	67	66	65	64
01	 1011	183	182	181	180
01	 0000	7	6	5	4
10	 1011	175	174	173	172
10	 1010	171	170	169	168
11	 0010	47	46	45	44
11	...	...	...	...	...

This cache resulted in a 4/12 hit rate.

Tricky Things to Remember:

- Check whether we are using byte addresses or word addresses
- Remember when calculating the number of bits needed to represent the index, we need to divide the number of blocks by the number of blocks per set (in a 2 way set associative cache, this is 2).

4-Way Set Associative Cache

Summary: A four-way set associative cache is a cache in which each index cooresponds to a set composed of four blocks. This means that there are four possible places for a block to be stored in the cache.

Example:

- Cache Details:
  - 8 blocks
  - 4 words per block
  - 32 bits per word (This is the default size we have been using for all of our examples)
  - Word Addresses:
    - 43
    - 171
    - 45
    - 178
    - 179
    - 5
    - 174
    - 46
    - 4
    - 181
    - 65
    - 180

1. **Convert to Binary Addresses:** The first thing we need to do is convert the word addresses into binary addresses. These addresses are going to be 30 bits long since each word is 32 bits long (and we need to subtract 2 bits for the word offset).

Word Address	Binary Address
43	.... 0010 1011
171	.... 1010 1011
45	.... 0010 1101
178	.... 1011 0010
179	.... 1011 0011
5	.... 0000 0101
174	.... 1010 1110
46	.... 0010 1110
4	.... 0000 0100
181	.... 1011 0101
65	.... 0100 0001
180	.... 1011 0100

   - **Note:** We are using word addresses, if we are using byte addresses, then we would need to multiply the word address by 4 (add 2 zeros to the end of the binary address).

2. **Decompose the Addresses:** In order to cache the addresses, we need to decompose them into their word number, index, and tag (from right to left).
   - **Word Number** $2^x =$ (Words per Block)
     - Since there are 4 words per block, we can determine the number of bits needed to represent the word number by taking the $2^x = 4$ and solving for x. This gives us $x = 2$.
   - **Index** $2^x =$ (Blocks in Cache)
     - Since there are 8 blocks in the cache (Divided into sets of 4), we can determine the number of bits needed to represent the index by taking $2^x = \frac{8}{4}$ and solving for x. This gives us $x = 1$.
   - **Tag** (Address Length) − (Word Number Length) − (Index Length)
     - Since the address length is 30 bits, the word number length is 2 bits, and the index length is 1 bits, we can determine the number of bits needed to represent the tag by taking $30 - 2 - 1 = 27$.

Number	Address	Tag	Index	Word
43	... 0000 0010 1011	... 00101	0	11
171	... 0000 1010 1011	... 10101	0	11
45	... 0000 0010 1101	... 00101	1	01

Number	Address	Tag	Index	Word
178	... 0000 1011 0010	... 10100	1	10
179	... 0000 1011 0011	... 10100	1	11
5	... 0000 0000 0101	... 00000	1	01
174	... 0000 1010 1110	... 10111	0	10
46	... 0000 0010 1110	... 00101	1	10
4	... 0000 0000 0100	... 00000	1	00
181	... 0000 1011 0101	... 10110	1	01
65	... 0000 0100 0001	... 01000	0	01
180	... 0000 1011 0100	... 10110	1	00

3. **Create the Cache:** Cache initial state:

Index	Tag	Word 3	Word 2	Word 1	Word 0
0	...	...	...	...	...
0	...	...	...	...	...
0	...	...	...	...	...
0	...	...	...	...	...
1	...	...	...	...	...
1	...	...	...	...	...
1	...	...	...	...	...
1	...	...	...	...	...

4. **Map the Addresses to the Cache:** For each address, we need to determine if it is a hit or a miss. We do this by checking both tags in the set that corresponds to the index of the address. If either of the tags matches the tag of the address, then it is a hit. If neither of the tags matches the tag of the address, then it is a miss. If both of the entries in the set are empty, then it is a miss.
5. **Update the Cache:** If the address is a miss, then we need to update the cache. We do this by replacing the word in the cache that corresponds to the index, tag, and word number of the address with the new entry. Then we need to update the rest of the words in the block with the surrounding words in memory.

   Final Cache State:

Index	Tag	Word 3	Word 2	Word 1	Word 0
0	... 00101	43	42	41	40
0	... 10101	171	170	169	168
0	... 10111	175	174	173	172
0	... 01000	67	66	65	64
1	... 00101	47	46	45	44
1	... 10100	179	178	177	176
1	... 00000	7	6	5	4
1	... 10110	183	182	181	180

This cache resulted in 4/12 hits.

Fully Associative Cache

Summary: A fully associative cache is a cache where each block can be placed in any location in the cache. This means that there is no index, and the tag is compared to all of the tags in the cache. This is the most flexible type of cache, but it is also the most expensive to implement.

Example:

- Cache Details:
  - 8 blocks
  - 4 words per block

- 32 bits per word (This is the default size we have been using for all of our examples)
- Word Addresses:
  - 43
  - 171
  - 45
  - 178
  - 179
  - 5
  - 174
  - 46
  - 4
  - 181
  - 65
  - 180

1. **Convert the Addresses to Binary:**
   - **Note:** We are using word addresses, if we are using byte addresses, then we would need to multiply the word address by 4 (add 2 zeros to the end of the binary address).

Number	Address
43	.... 0010 1011
171	.... 1010 1011
45	.... 0010 1101
178	.... 1011 0010
179	.... 1011 0011
5	.... 0000 0101
174	.... 1010 1110
46	.... 0010 1110
4	.... 0000 0100
181	.... 1011 0101
65	.... 0100 0001
180	.... 1011 0100

2. **Decompose the Addresses:** In order to cache the addresses, we need to decompose them into their word number, index, and tag (from right to left).
   - **Word Number** $2^x =$ (Words per Block)
     - Since there are 4 words per block, we can determine the number of bits needed to represent the word number by taking the $2^x = 4$ and solving for x. This gives us $x = 2$.
   - **Index** 0
     - In a fully associative cache, there is no index, so we can set the number of index bits to 0
   - **Tag** (Address Length) − (Word Number Length) − (Index Length)
     - Since the address length is 30 bits, the word number length is 2 bits, and the index length is 1 bits, we can determine the number of bits needed to represent the tag by taking $30 - 2 - 1 = 27$.

Number	Address	Tag	Word
43	... 0000 0010 1011	... 001010	11
171	... 0000 1010 1011	... 101010	11
45	... 0000 0010 1101	... 001011	01
178	... 0000 1011 0010	... 101001	10
179	... 0000 1011 0011	... 101001	11
5	... 0000 0000 0101	... 000001	01
174	... 0000 1010 1110	... 101110	10
46	... 0000 0010 1110	... 001011	10
4	... 0000 0000 0100	... 000001	00
181	... 0000 1011 0101	... 101101	01
65	... 0000 0100 0001	... 010000	01
180	... 0000 1011 0100	... 101101	00

3. **Map the Addresses to the Cache:** For each address, we need to determine if it is a hit or a miss. We do this by checking to see if the tag of the address matches the tag of any of the entries in the cache. If the tag matches, then it is a hit. If the tag does not match, then it is a miss. If all of the entries in the cache are empty, then it is a miss.

4. **Update the Cache:** If the address is a miss, then we need to update the cache. We do this by replacing the entry in the cache with the new entry. Then we need to update the rest of the words in the block with the surrounding words in memory.

   Final Cache State:

Tag	Word 3	Word 2	Word 1	Word 0
... 101010	171	170	169	168
... 101001	179	178	177	176
... 101110	175	174	173	172
... 001011	47	46	45	44
... 000001	7	6	5	4
... 101101	183	182	181	180
... 010000	67	66	65	64

   This cache resulted in 4/12 hits.
