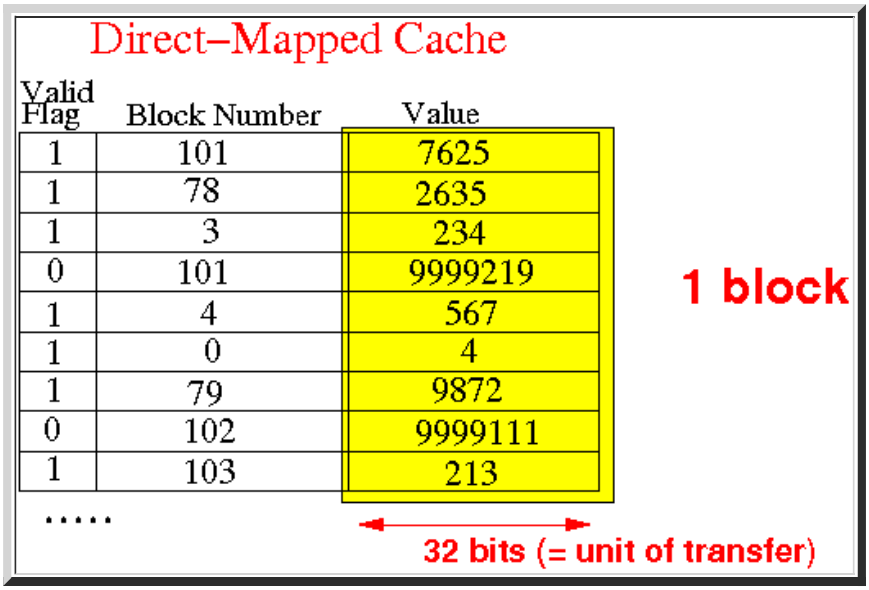
Direct-Mapped Cache and its architecture

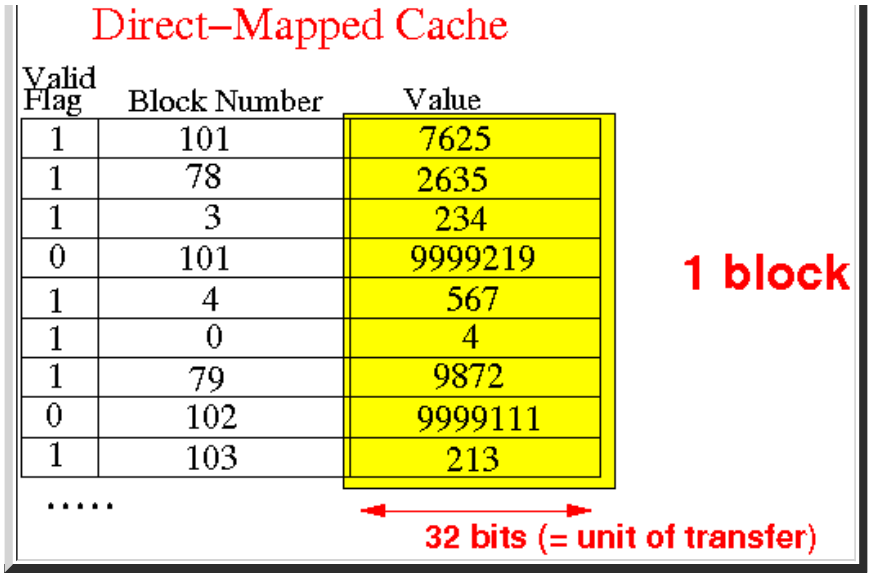
- **Blocks in a Direct-Mapped cache**
  - **Block** in a **Direct-Mapped cache**:
    - **One block** = the ***whole*** **cache** !!!

    Example:

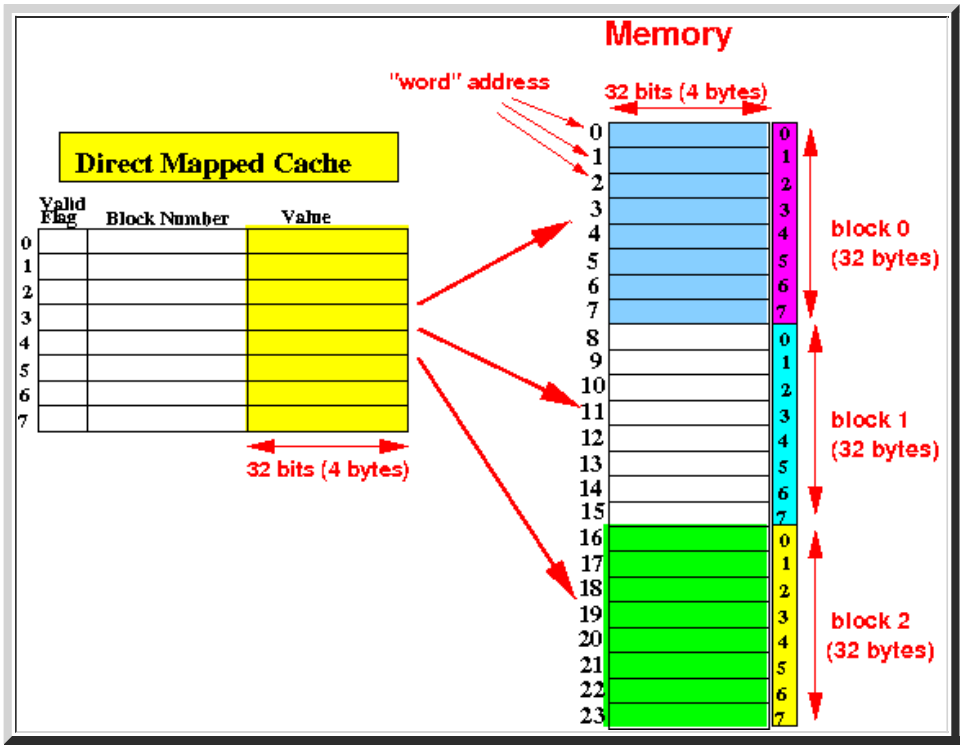


- **Partitioning of the memory into block in Direct-Mapped cache**
  - The **memory** is **partitioned** into:
    - **block size** that are **equal** to the **size** of the **Direct-mapped cache**.

  - **Example:**
    - **Suppose** we have a **Direct-Mapped cache** with **8 entries** (**each entry** contains an **32 bits (4 bytes) value**:



Then the **memory** is **partitioned** in the **following manner**:



Note:

- A **block** has **8 entries**
- **Each entry** has **4 bytes**

- **So:**
  - **Each block** has **8 × 4 = 32 bytes** !!!

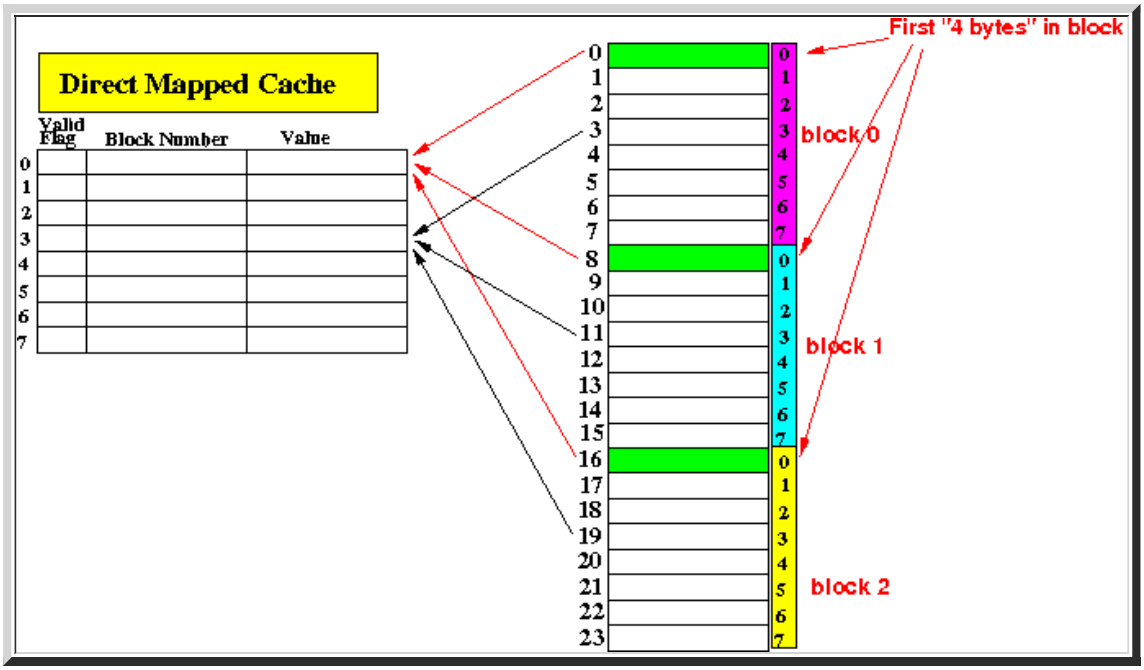
- The ***first* 32 bytes** is **block 0**
- The ***second* 32 bytes** is **block 1**

- **And so on**.

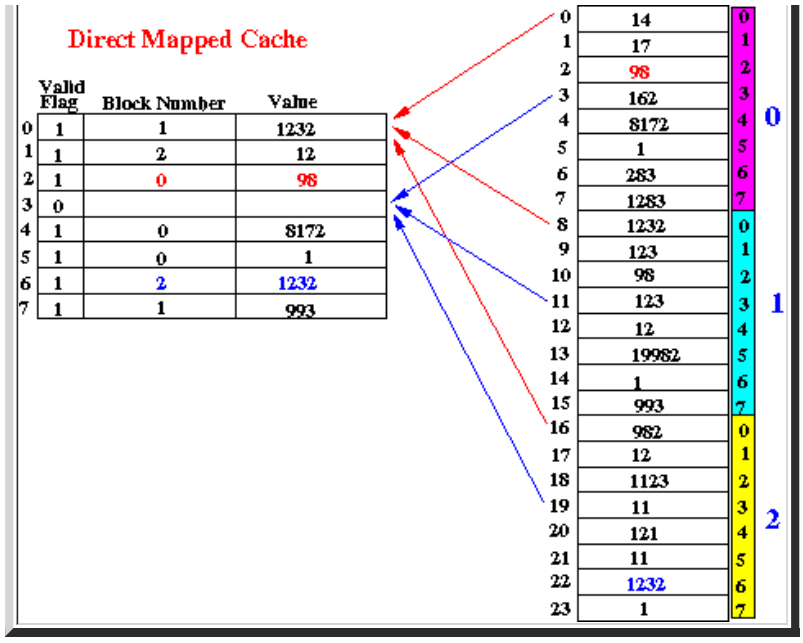
- **Restriction (rule) to cache data from memory in a Direct-Mapped cache**
  - **Restriction (rule)**:

    - The ***first*** **transfer unit (= 32 bits, or 4 bytes)** in ***any*** **block** **must** be:
      - **cached** in the ***first*** **entry** in the **Direct-mapped cache**
    - The ***second*** **transfer unit (= 32 bits, or 4 bytes)** in ***any*** **block** **must** be:
      - **cached** in the ***second*** **entry** in the **Direct-mapped cache**
    - **And so on**

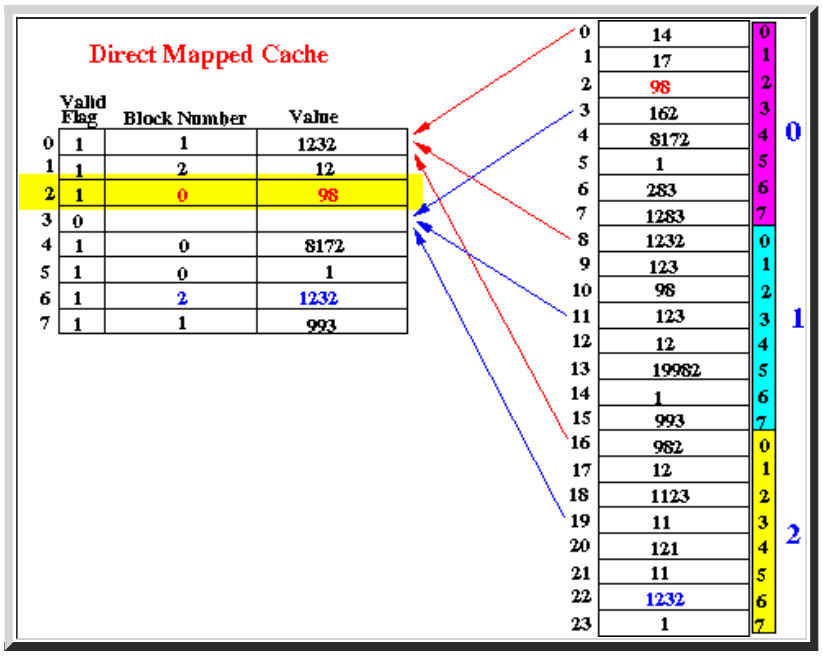
    Graphically:



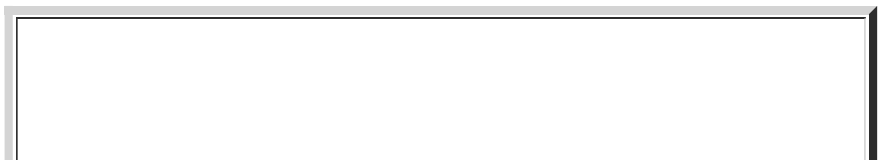
- **Example content of a Direct-Mapped Cache**
  - **Example** of the **content** of an **Direct-Mapped cache**:

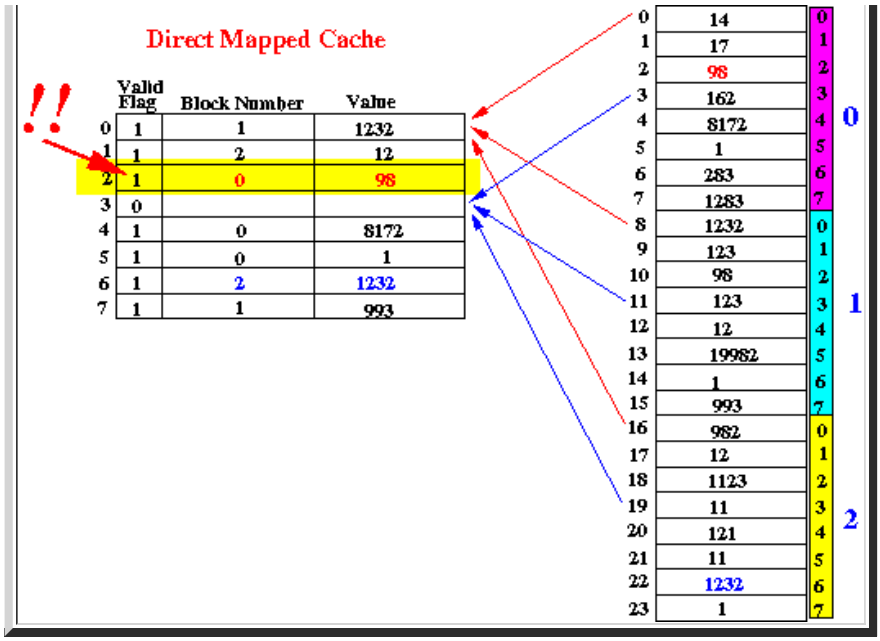


- **"Decoding" the information stored in an Associative Cache**
  - **Example** of **decoding** the **information** stored in an **Associative Cache**:
    - Consider the **highlighted entry**:



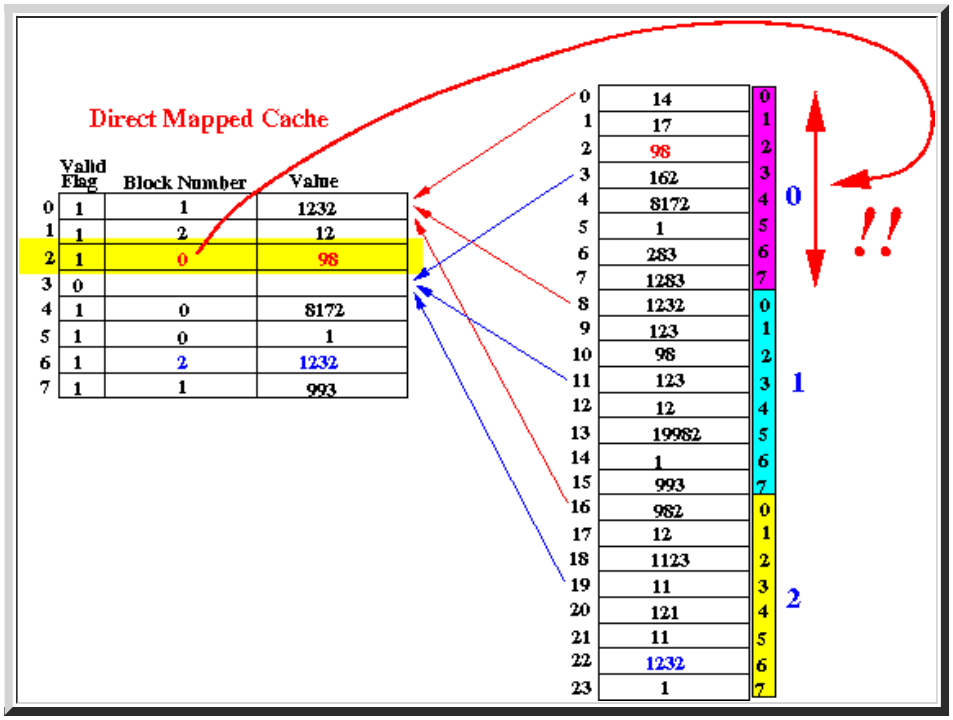
    - The **entry** in the **slot** is **valid**:



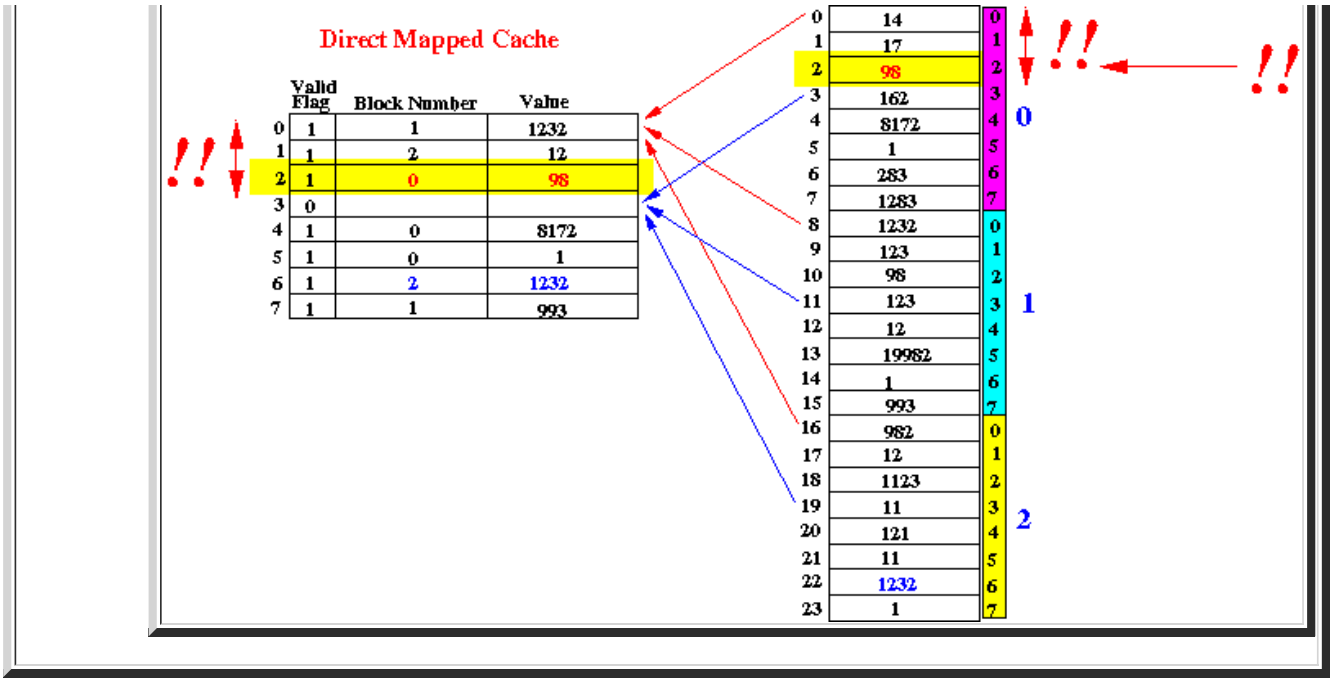


So it **contains** ***correct*** **value** stored in **memory**

- The **block number** is **block 0** in **memory**:

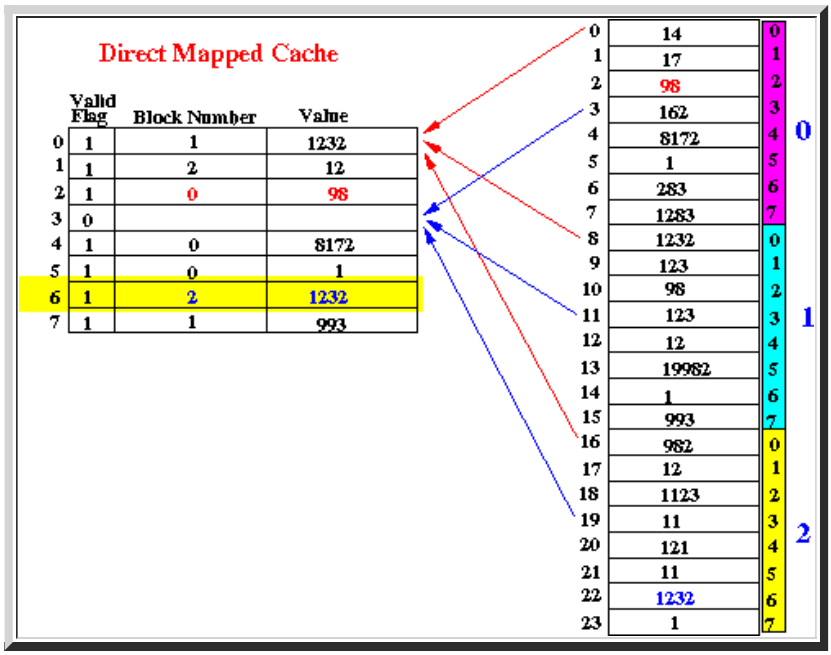


- **According** to the **Direct-Mapped** ***Rule***, the **entry must** be ***this*** **one**:

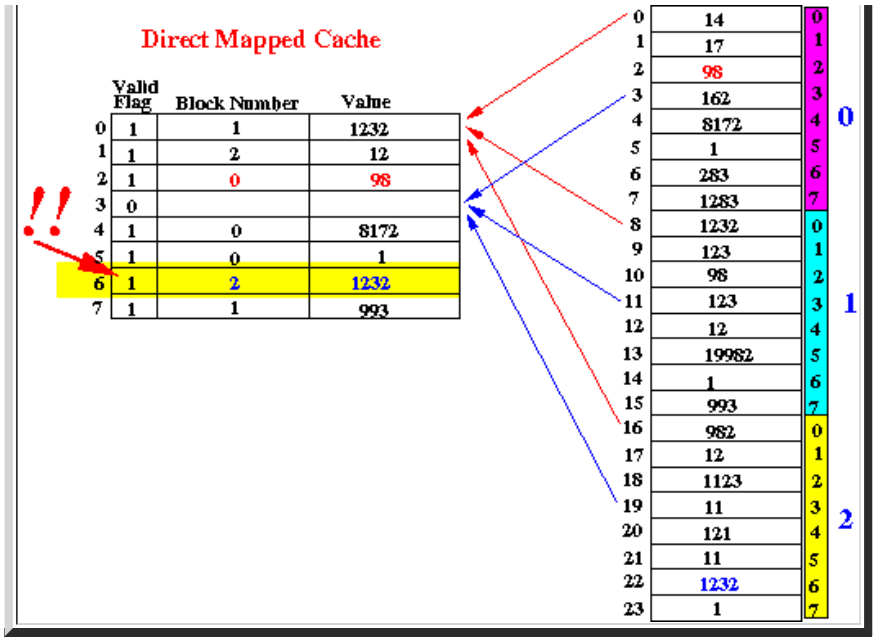


- **Another example** of **decoding** the **information** stored in an **Associative Cache**:

  - Consider the **highlighted entry**:

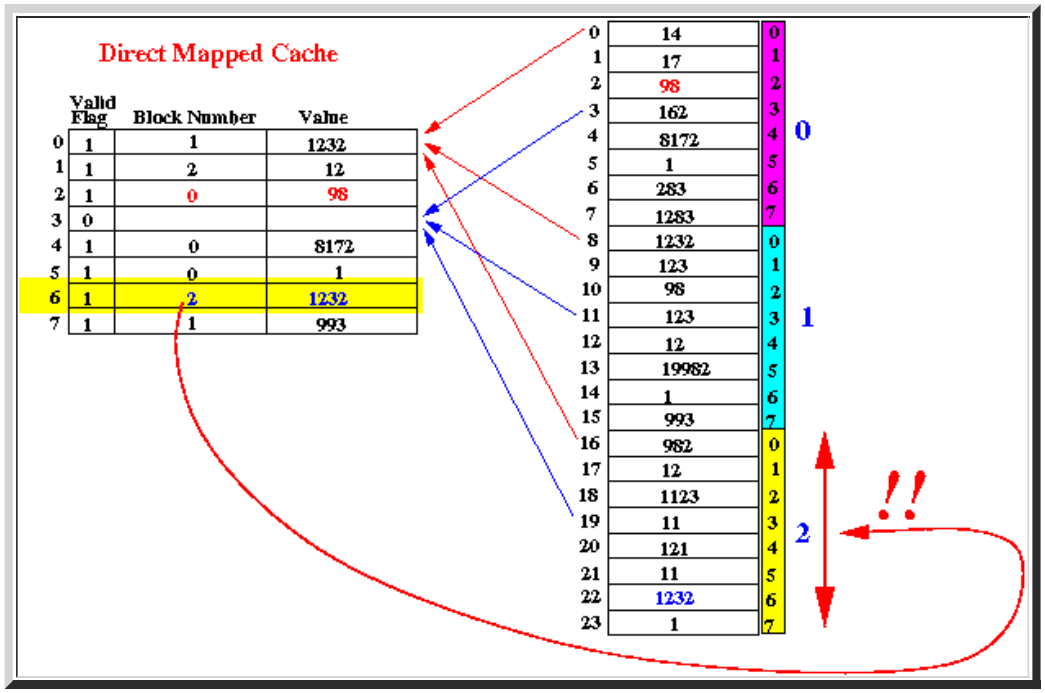


  - The **entry** in the **slot** is **valid**:

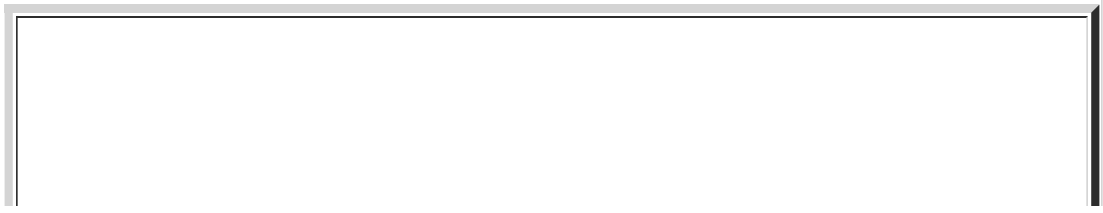


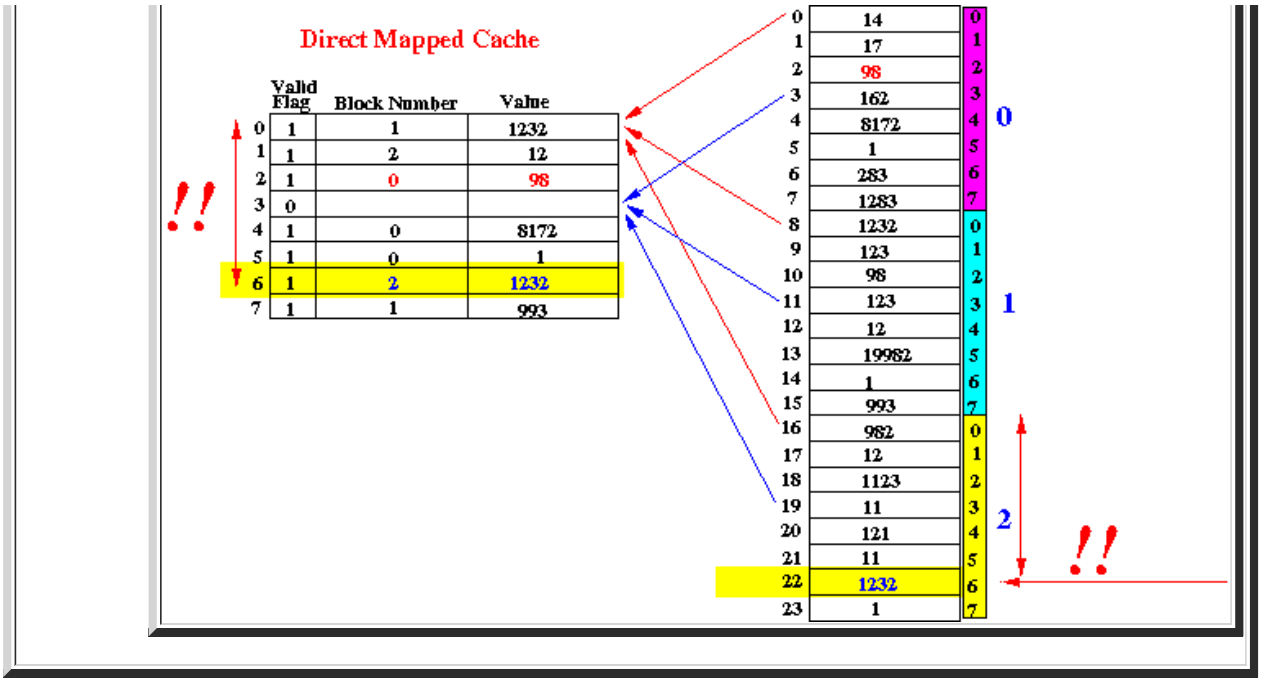
So it **contains *correct* value** stored in **memory**

- The **block number** is **block 2** in **memory**:

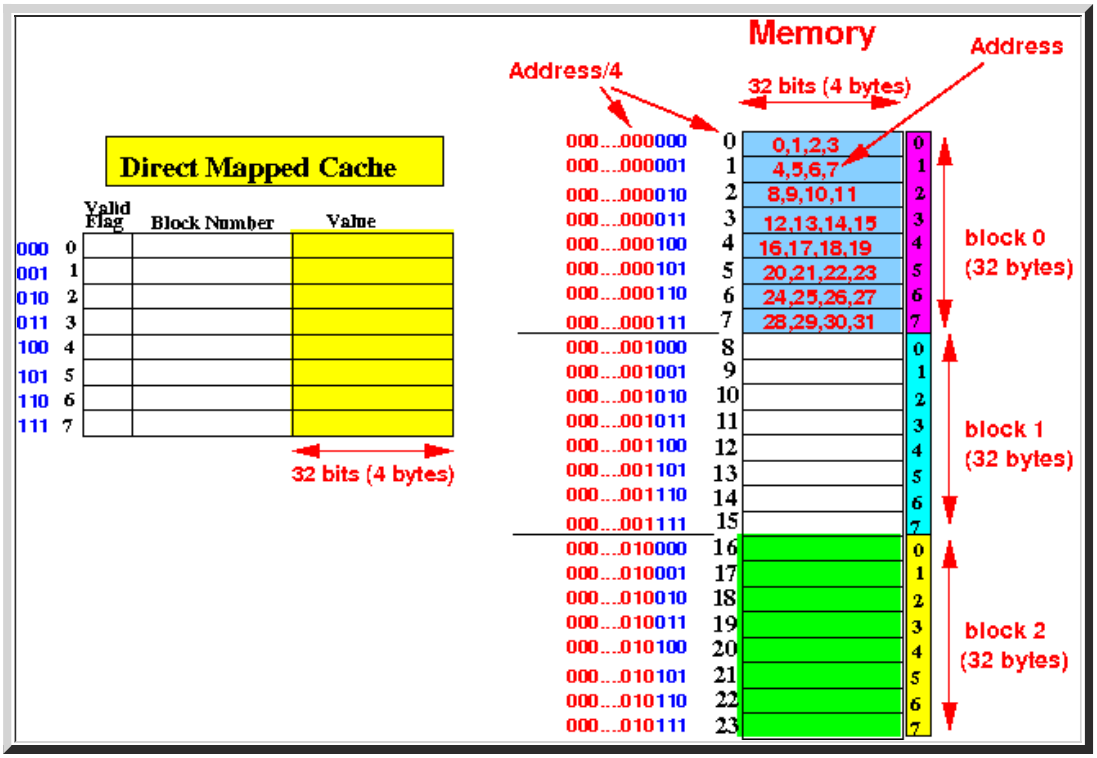


- **According** to the **Direct-Mapped *Rule***, the **entry must** be ***this* one**:





- **How to map an address to a cache entry in Direct Mapped cache**
  - You need to write the **addresses** out in **binary** to see the **association**:



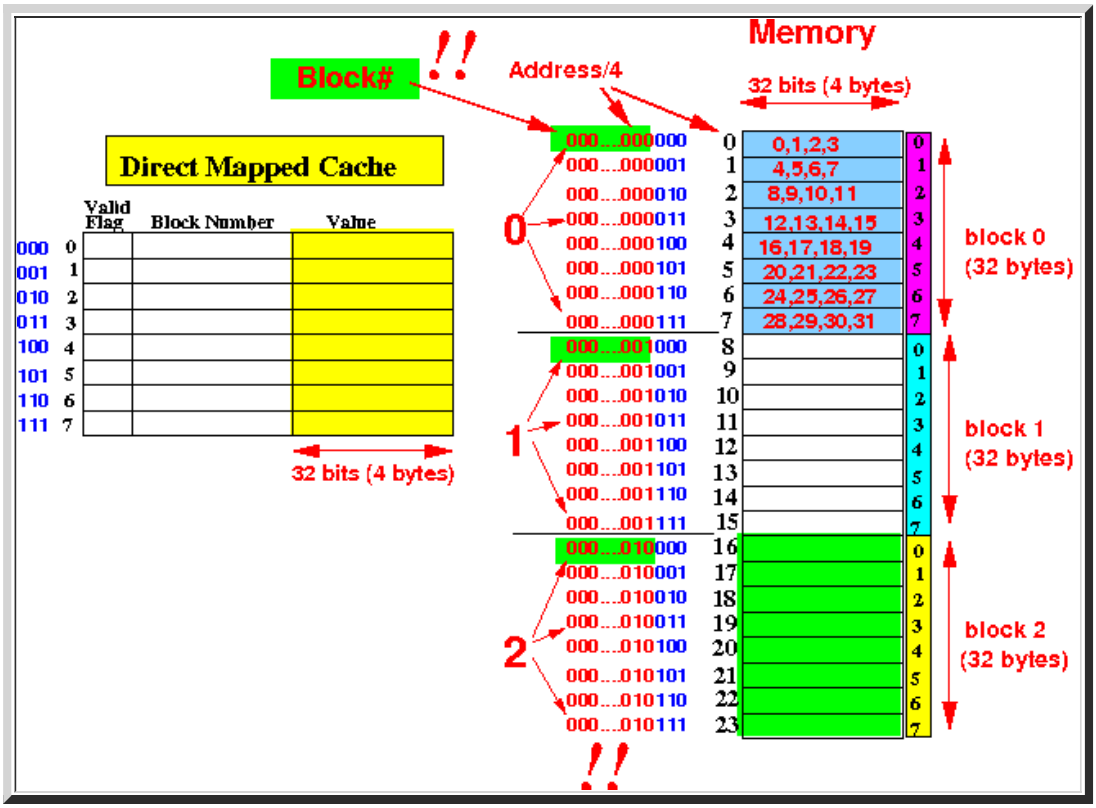
Observation:

- Since we **cache 4 bytes** in a **cache entry**, we need to **divide** the **address** by **4** to obtain a **"word" address** first

- I have **written** out the `Address/4` values **both** in **binary** and **decimal** in the **above figure**

  (You can find the `Address/4` values at the **center** of the **figure**)

- You can see the **block number** is **here**:

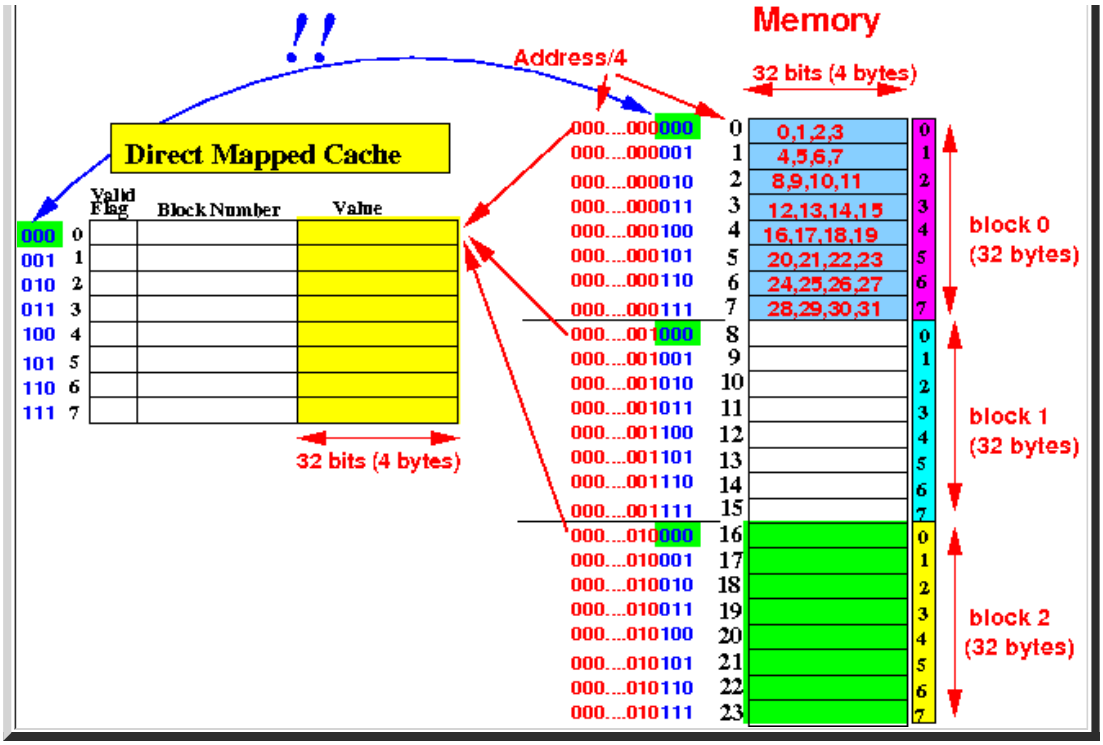


Conclussion:

- *If* we have **8 (= 2^3) entries**, **then**:
  - **Block number** = **(Address/4) / 8**

- **Furthermore**, the ***last* 3 bits** of the **value** `"Address/4"` (***if*** we have **8 (= 2^3) entries** in the **cache**) **indicates**:
  - The ***entry*** in the **Direct-Mapped cache** used to **stored** the **value**

Graphically:



Conclusion:

- *If* we have **8 (= 2^3) entries**, **then**:
  - **Block number** = **(Address/4) % 8**

- **Using the Direct-Mapped cache --- with (memory) addresses**
  - **Suppose** the **content** of an **Direct-Mapped Cache** is:

Direct Mapped Cache

	Valid Flag	Block Number	Value
0	1	1	1232
1	1	2	12
2	1	0	98
3	0		
4	1	0	8172
5	1	0	1
6	1	2	1232
7	1	1	993

Note:

- The **size** of a **block** = **8 (entries) × 4 = 32 bytes** !!!

- The CPU now makes a read request for the byte at memory address 10

  Question:

  - Is this byte found in the cache ?

- **Answer:**

  - We first compute the "word" address:

    ```
    "Word" address = Byte address/4
                   = 10/4
                   = 2
    ```

  - The block# is equal to:

    ```
    2/8 = 0      (because we have 8 entries in cache)
    ```

  - The location of the entry in the cache used to store the value is:

    ```
    Location = 2 % 8 = 2
    ```

  - Now look at the location 2 in the cache:

    Direct Mapped Cache

	Valid Flag	Block Number	Value
0	1	1	1232
1	1	2	12
2	1	0	98
3	0		
4	1	0	8172
5	1	0	1
6	1	2	1232
7	1	1	993

    block#

    At location 2 we find the block # = 0 (and entry is valid), so:

    - The value is stored in the cache !!!

- **Summary on how to use an associative cache**

  - **Summary**

- **CPU** makes a **read request** for the **content** at some **memory location (memory address)**

- We **first** compute the **"word" address**:

```
"word" address = (byte) address / 4
```

- We **compute** the **location**:

```
Location = "word" address % n  (if we have n entries in cache)
```

- We **also compute** the **block#**:

```
Block# = "word" address / n  (if we have n entries in cache)
```

- If the **entry** at `Location` in the **cache** contains the `Block#` (and the **entry** if **valid**), then:

  - We have a **cache hit**

  Otherwise:

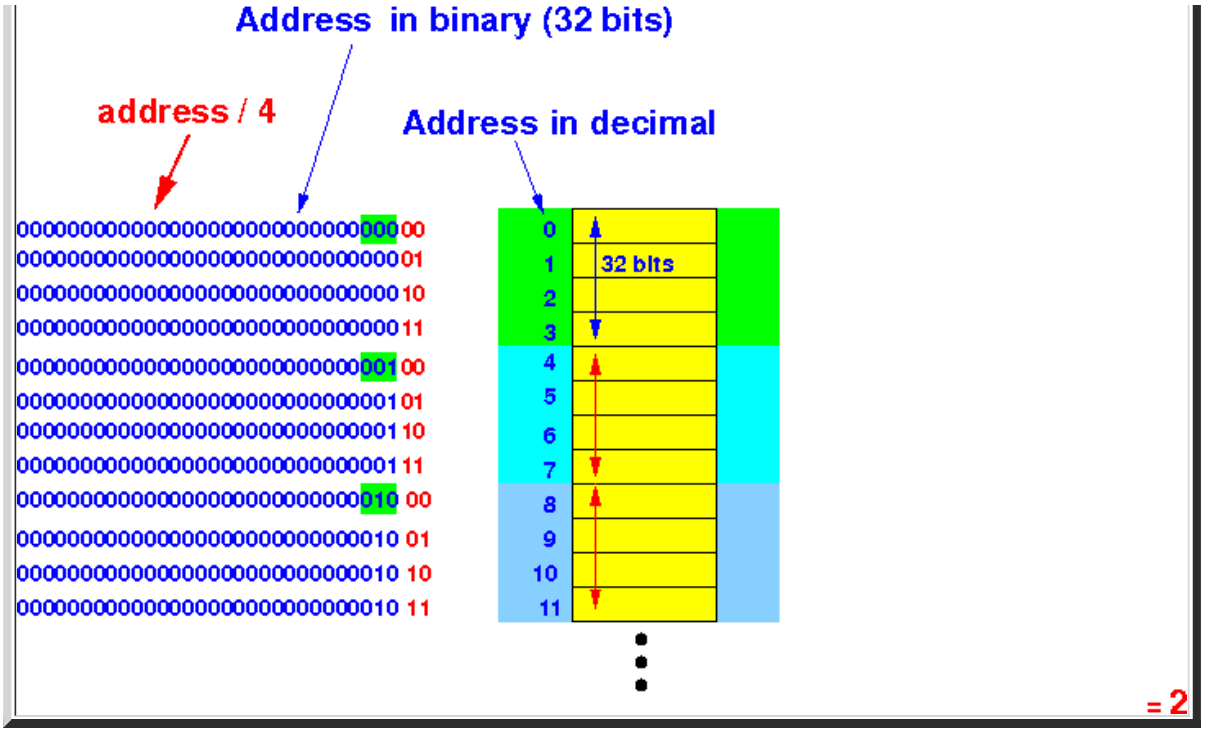
  - We have a **cache *miss***

- **Prelim: how to find the block number and the Location from an address**
  - **Fact:**

    - **Computer addresse**s are expressed in **binary** inside the **computer**

      (Each **bit** is **carried** by a **copper wire** on the **computer mother board**)

  - The **following figure** shows:



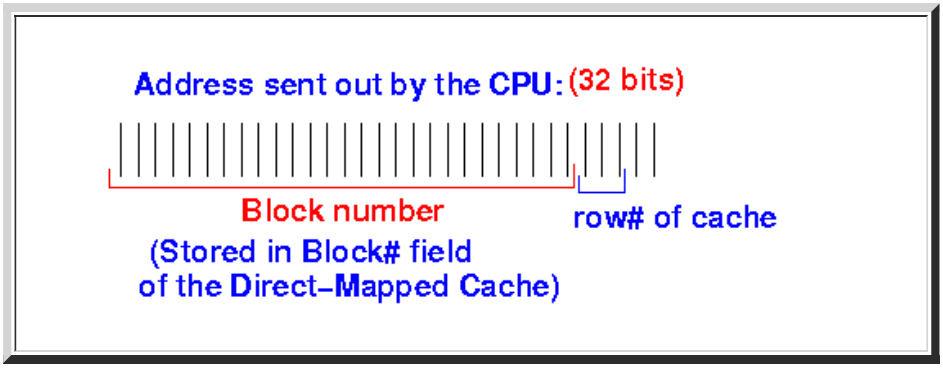
that:

- **Block number** is **obtained** from a **(memory) address** by:
  - **Taking away** the ***last*** **(2+3) bits** from the **memory address**

  2 bits to get the value **address/4**
  Then **3** more bits because there are **$8 = 2^3$ entries** in the **cache**

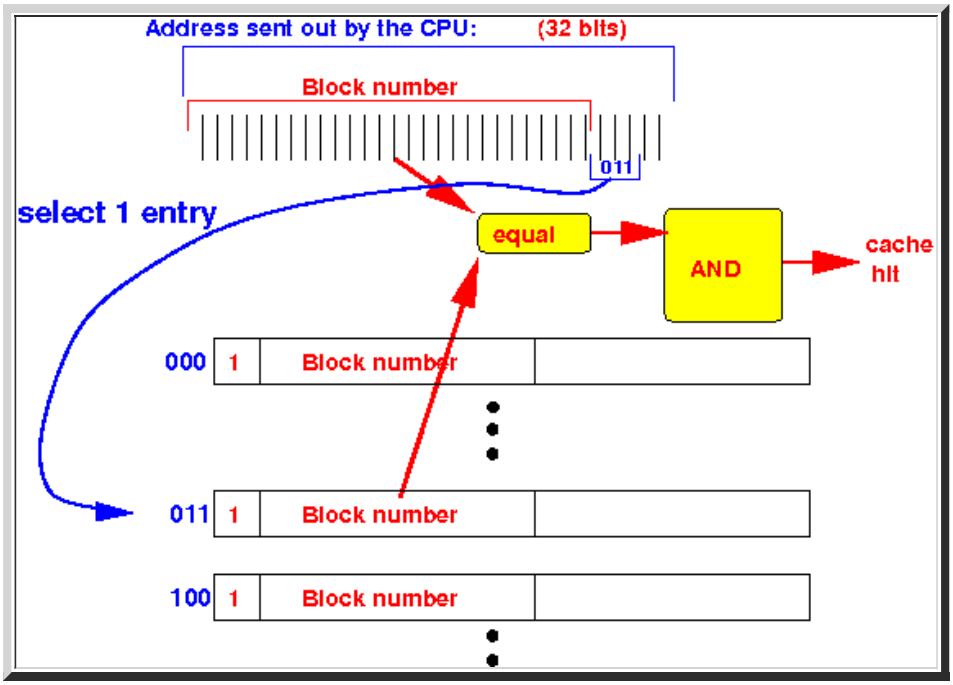
- **Location** used in the **Direct-Mapped cache** = **bit positions 5,4,3** from the **end**

- **Therefore:**



- **Circuitry to check if one entry in Associative Cache contains a cache hit**
  - The **following diagram** is the **logic** used to **check** if an **arbitrary entry** in an **Direct-Mapped cache** contains the **requested**

value:



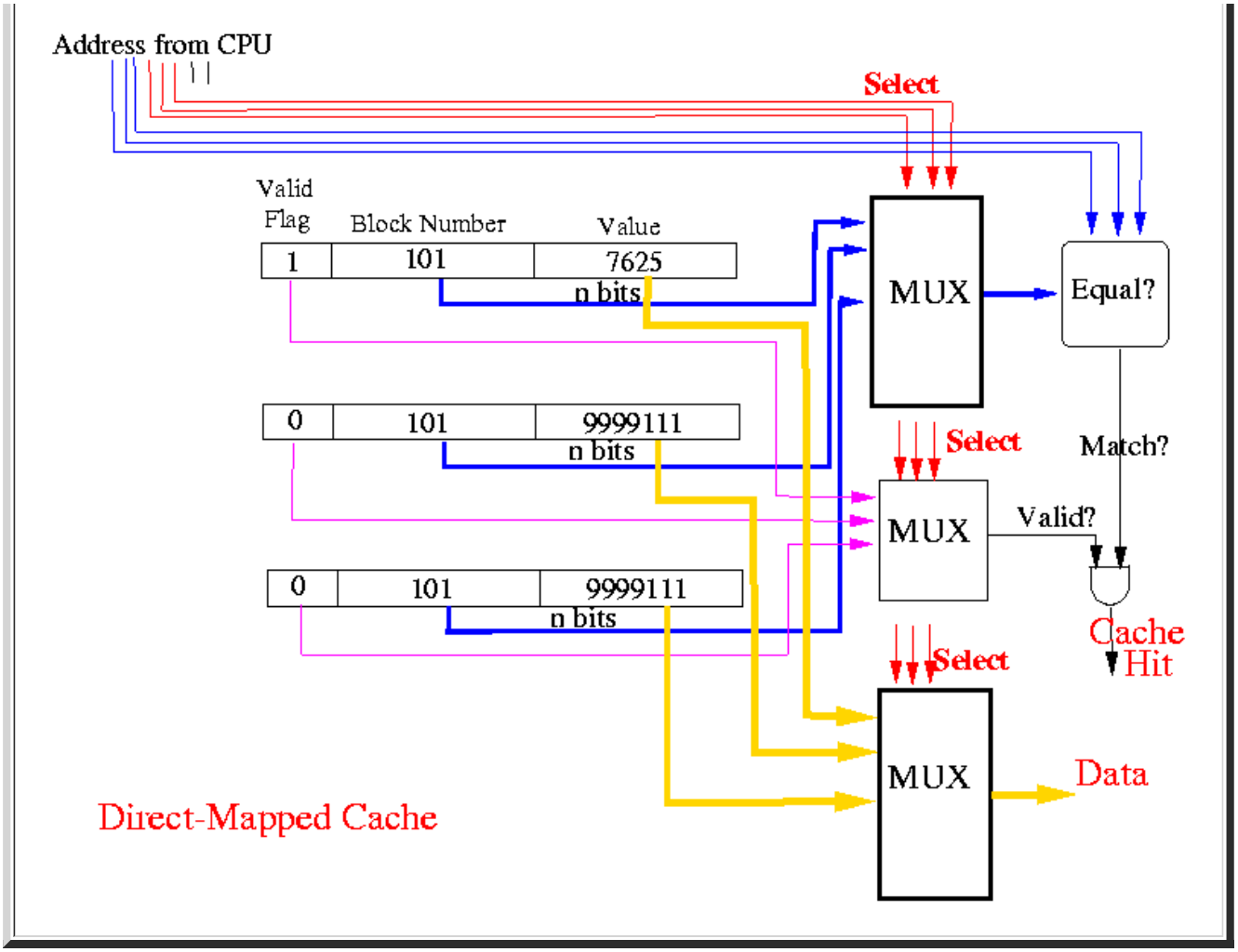
Depicted logic:

- The **block number** requested by the **CPU** is **equal** to the **block number** stored in the **cache entry** at the ***location*** given

---

- **And** the **cache entry** must be ***valid***

- **Architecture of an Associative Cache**
  - The **architecture** of an **Associative cache** is as **follows**:



Explanation:

- **Each MUX** will **select**:
  - **Valid bit**
  - **Block number**
  - Value

  at the `Location` entry in the **Direct-Mapped cache**

- The **Block#** at the `Location` entry in the **Direct-Mapped cache** is **compared** to the **block#** requested by the **CPU**

- If these **block#'s** are **equal** and the **valid bit** is **1**, we have a **cache hit**

- **Strength and weakness of the Direct-Mapped Cache**
  - **Strength:** ***cheap***
    - The **Direct-Mapped cache** uses ***only*** uses **1 compare circuit** !!!

- **Weakness: inflexible**

  - A **given word (= 4 bytes)** in **memory** ***must*** be **stored** in ***one*** **specific entry** in the **Direct-Mapped cache**