香港中文大學
The Chinese University of Hong Kong

*CSCI2510 Computer Organization*

# Tutorial 08: Direct Mapping Implementation

**Tsun-Yu YANG**

*yangty@cse.cuhk.edu.hk*

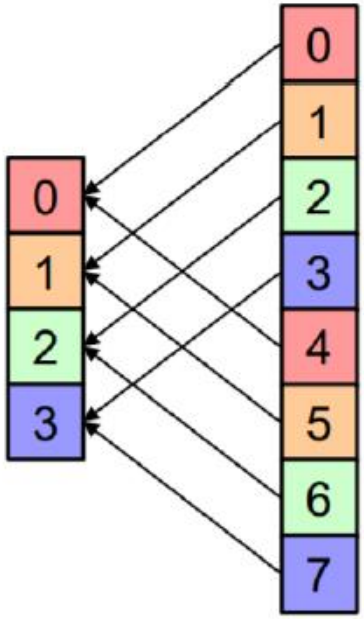
# Outline

- Review of Direct Mapping
- Advanced MASM Instructions
- Implementation of Direct Mapping with MASM Code

# Review of Direct Mapping

## Direct

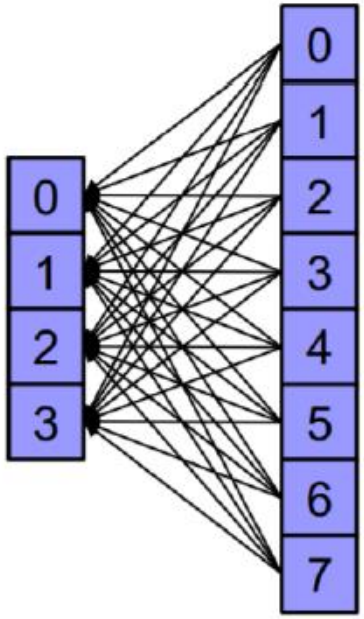
A Memory Block is **directly mapped** (%) to a Cache Block.




## Associative

A Memory Block can be **mapped to any** Cache Block.

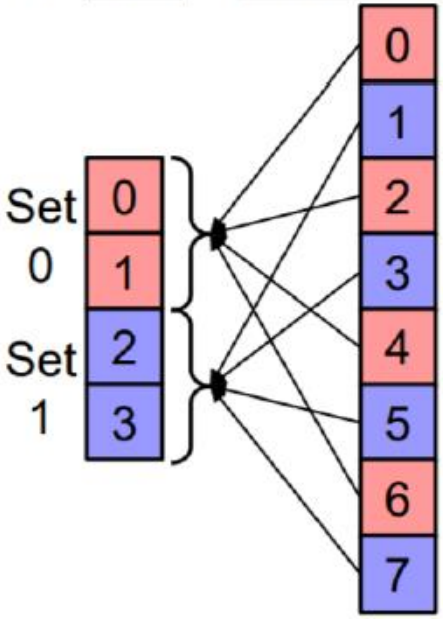
(First come first serve!)




## Set Associative

A Memory Block is **directly mapped** (%) to a **Cache Set**.

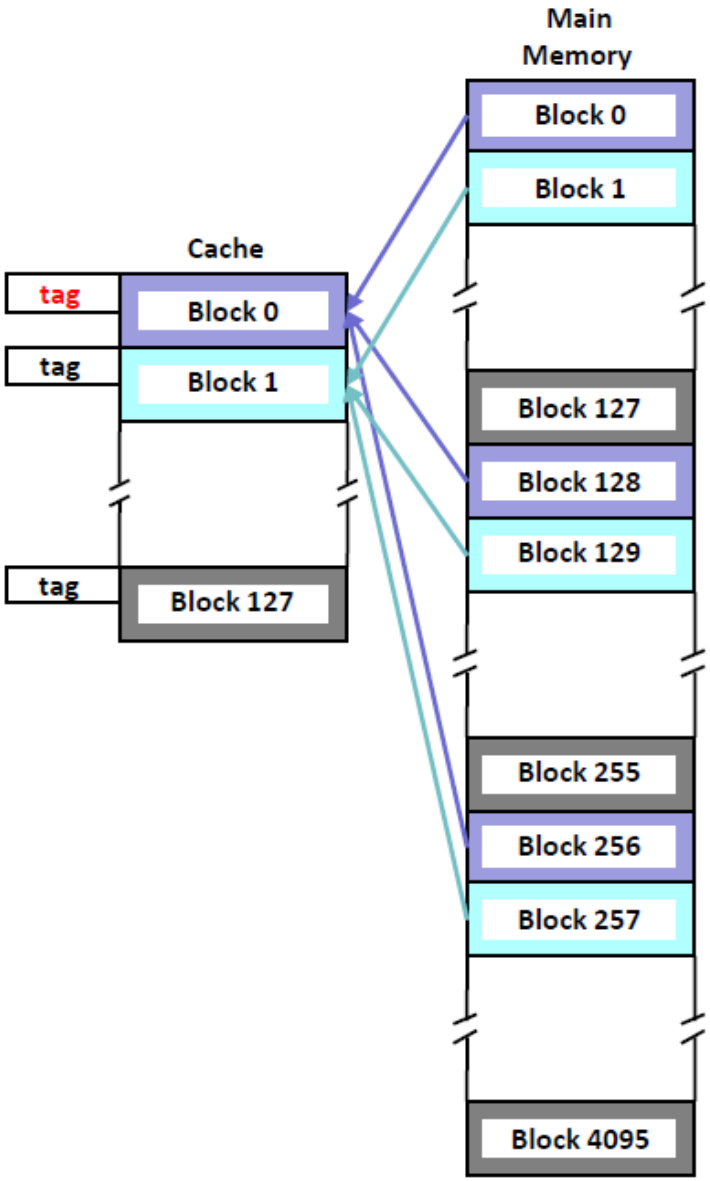
In a **Set**? **Associative**



# Review of Direct Mapping

- **Direct Mapped Cache**:
  - Each Memory Block (MB) will be directly mapped to a Cache Block (CB)

- **Direct Mapping Function**:
  - MB #$j$ → CB #($j$ mod 128)
  - 128 is because there are 128 CBs in Cache
  - A tag is needed for each CB
    - A CB is shared by many MBs, tag is used for clarifying which MB is stored



# Find Mapped CB w/ idiv Instr.

- To implement MB #$j$ → CB #($j$ mod 128) in MASM
  - We need the help of **idiv** instruction
    - **Idiv** instruction executes signed division, and produce quotient and remainder

- Given a memory block ID $x$, below shows the MASM code to find the corresponding mapped cache block:

```
mov eax, x
mov edx, 0 ; edx:eax is the Dividend
idiv 128 ; 128 is the Divisor,
         ; which is # cache blocks
; eax stores Quotient
; edx stores Remainder -> the Mapped Cache Block
```

# Problems of idiv Instruction

- For executing every idiv instruction:
  - It needs to use 2 registers, EAX and EDX
  - It takes too long to finish the task

| Instr. | ADD | SUB | IMUL | IDIV |
|---|---|---|---|---|
| CPU clocks | 2 | 2 | 9-38 | 43 |

- We can use advanced bit-wise instructions to do the division under some certain conditions
  - The value of divisor needs to be the power of 2

# Advanced MASM Instructions

- Fortunately, the number of cache blocks is usually the power of 2 ($2^n$)
  - We can use the much faster bit-wise instructions to replace the expensive idiv instruction

| Instr. | AND | OR | XOR | NOT | SHR | SHL |
|---|---|---|---|---|---|---|
| CPU clocks | 2 | 2 | 2 | 2 | 3 | 3 |

.....

# Bit-wise Instruction: AND & OR

- **AND** operand1, operand2
  - Bit-wise AND operation
  - Truth Table:

| a | b | out |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

MSB LSB

0 0 1 0 1 1 0 (operand1)

AND 0 0 1 0 0 1 1 (operand2)

0 0 1 0 0 1 0 (operand1)

- **OR** operand1, operand2
  - Bit-wise OR operation
  - Truth Table:

| a | b | out |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

MSB LSB

0 0 1 0 1 1 0 (operand1)

OR 0 0 1 0 0 1 1 (operand2)

0 0 1 0 1 1 1 (operand1)

# Useful Examples of AND & OR

- **Q1**: For $x = (0101\ 1100)_2$, we want the top 4 bits to be 0, and the bottom 4 bits remain their values

| | | |
|---|---|---|
| | 0 1 0 1 1 1 0 0 | $x$ |
| AND | 0 0 0 0 1 1 1 1 | *Mask* |
| | 0 0 0 0 1 1 0 0 | → *Top 4 bits are masked to be 0* |

- **Q2**: For $x = (0101\ 1100)_2$, we want the top 4 bits to be 1, and the bottom 4 bits remain their values.

| | | |
|---|---|---|
| | 0 1 0 1 1 1 0 0 | $x$ |
| OR | 1 1 1 1 0 0 0 0 | *Mask* |
| | 1 1 1 1 1 1 0 0 | → *Top 4 bits are masked to be 1* |

# Bit-wise Instruction: XOR & NOT

- **XOR** operand1, operand2
  - Bit-wise XOR operation
  - Truth Table:

| a | b | out |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

MSB LSB

0 0 1 0 1 1 0 (operand1)

XOR 0 0 1 0 0 1 1 (operand2)

0 0 0 0 1 0 1 (operand1)

- **NOT** operand1
  - Bit-wise NOT operation
  - Truth Table:

| in | out |
|---|---|
| 0 | 1 |
| 1 | 0 |

MSB LSB

0 0 1 0 1 1 0 (operand1)

NOT

1 1 0 1 0 0 1 (operand1)

# Bit-wise Instruction: SHR & SHL

- **SHR** operand1, operand2
  - Shift the bit pattern in operand1 to right by the operand2 amount of bits

MSB LSB

0 0 1 0 1 1 0 (operand1)

SHR **2** (operand2)

0 0 0 0 1 0 1 (operand1)

---

- **SHL** operand1, operand2
  - Shift the bit pattern in operand1 to left by the operand2 amount of bits

MSB LSB

0 0 1 0 1 1 0 (operand1)

SHL **2** (operand2)

1 0 1 1 0 0 0 (operand1)

# Useful Examples of SHR and SHL

- **Q1**: For $x = (0000\ 1100)_2$, we want to divide $x$ by the value $4\ (= 2^2)$

$$x = 12 \rightarrow \frac{x}{4} = 3$$

Right shift 1 bit → divide by 2

MSB LSB

0 0 0 0 1 1 0 0 (= 12)

SHR 2

0 0 0 0 0 0 1 1 (= 3)

- **Q2**: For $x = (0000\ 1100)_2$, we want to multiply $x$ by the value $4\ (= 2^2)$

$$x = 12 \rightarrow 4x = 48$$

Left shift 1 bit → multiply by 2

MSB LSB

0 0 0 0 1 1 0 0 (= 12)

SHL 2

0 0 1 1 0 0 0 0 (= 48)

# Notes of Advanced MASM Instr.

- For simplicity, previous slides show the examples of 8-bit data operation.
  - But in reality, a register is 32-bit

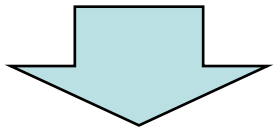
- Before using the bit-wise instructions, please be familiar with the binary representation of a value

- The bit-wise instructions are very useful in various of situations
  - Question: How to use **AND** to know whether a number is divisible by 2? (In general way, use idiv → check *remainder*)

# Find Mapped CB w/ Bit-wise Instr.

- Given a memory block ID $x$, below shows the MASM code to find the corresponding mapped cache block:

```
mov eax, x
mov edx, 0 ; edx:eax is the Dividend
idiv 128 ; 128 is the Divisor,
         ; which is # cache blocks
; eax stores Quotient
; edx stores Remainder -> the Mapped Cache Block
```



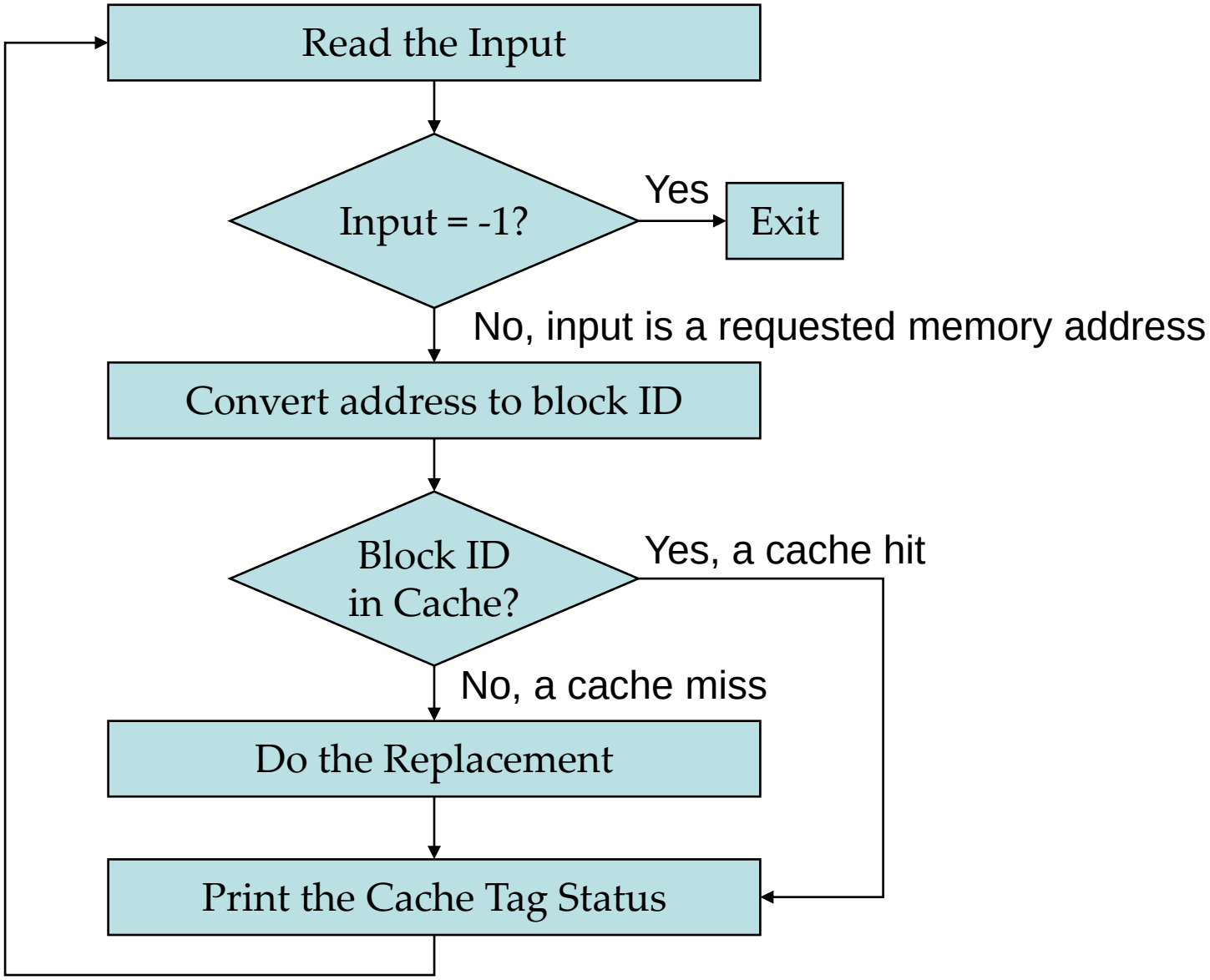
```
mov eax, x
and eax, 127 ; 128 = 0..010000000
; eax now stores the Reminder of (x / 128)
```

# How to use AND to find the Remainder

- Let's consider the binary to decimal conversion
- Given $x = 0010\ 1101_2 = 2^5 + 2^3 + 2^2 + 2^0$
- Given the divisor $y = 0000\ 1000_2 = 2^3$
  - The divisor must be in the form of power of 2
- $\frac{x}{y} = \frac{2^5+2^3+2^2+2^0}{2^3} = \frac{2^5+2^3}{2^3} + \frac{2^2+2^0}{2^3}$ ⇨ $00101101_2$ (The remainder!)
- $00101101_2$ AND $00000111_2 = 00000101_2$ = Remainder

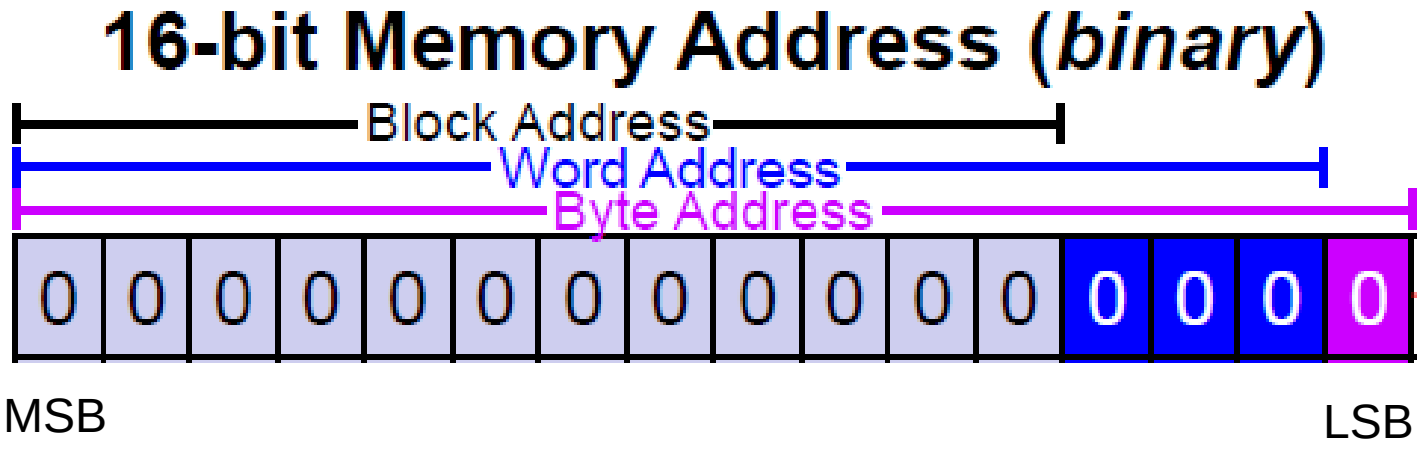
  $x$ $\quad$ $y - 1 = (2^3 - 1)$

# Flowchart of Direct Mapping Program



# Implementation of Direct Mapping

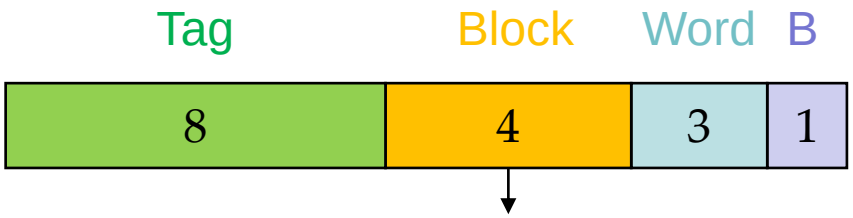
- In this example, we use the same configuration as the one in the lecture:
  - Memory address is 16-bit: the input value < $2^{16}$ = 65536
  - The word size is 2 bytes
  - Each block contains $2^3$ words = 16 bytes = $2^4$ bytes



# Implementation of Direct Mapping

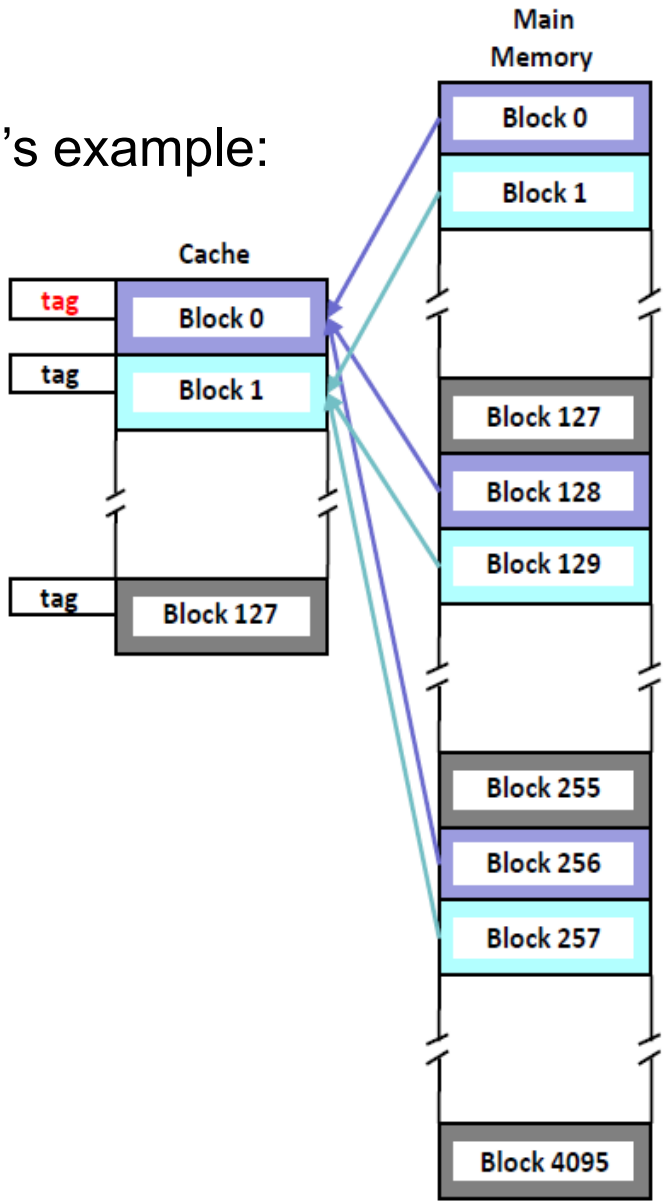
- Suppose we have 16 CBs

Our example:




4-bit can represent 0~15,
which is enough for all CBs

Lecture's example:




- **Please note that since we cannot directly manage CPU cache, we allocate a memory space to simulate cache for the sake of practice**
- Let's see the code directly and explain the code line by line!

# Summary

- Review of Direct Mapping
- Advanced MASM Instructions
- Implementation of Direct Mapping with MASM Code