How to Compute Matrix Multiplication?

Liyuan Xu

Matrix Multiplication

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \qquad B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

$$C = AB \qquad c_{i,j} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

How to implement it?

Two Codes

Code1

```
for (i = 0; i < MAT_SIZE; i++){
    for (j = 0; j < MAT_SIZE; j++){
        for (k = 0; k < MAT_SIZE; k++){
            c[i][j]+=a[i][k]*b[k][j];
        }
    }
}
```

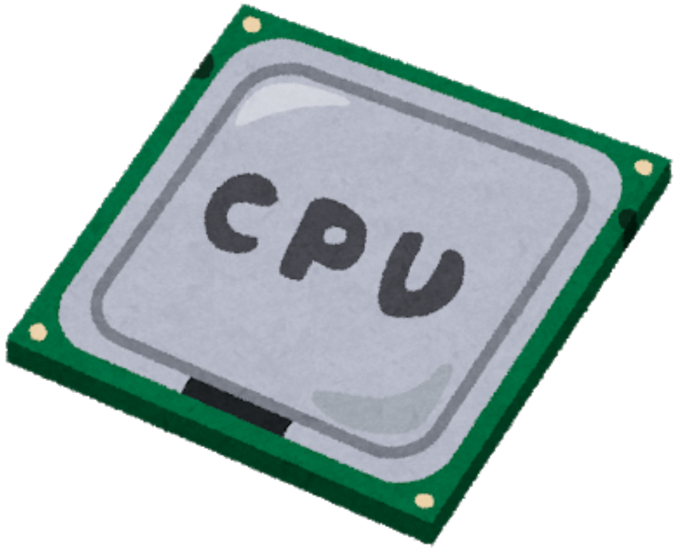
128 [sec] (N=2000)

Code2

```
for (i = 0; i < MAT_SIZE; i++){
    for (k = 0; k < MAT_SIZE; k++){
        for (j = 0; j < MAT_SIZE; j++){
            c[i][j]+=a[i][k]*b[k][j];
        }
    }
}
```

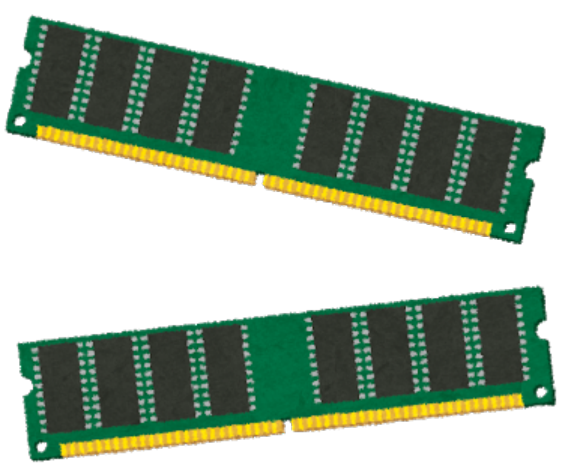
22[sec] (N=2000)

Computer Architecture (My computer)



CPU

- Process arithmetic operation
- Clock frequency: ≒2.7 GHz



Memory

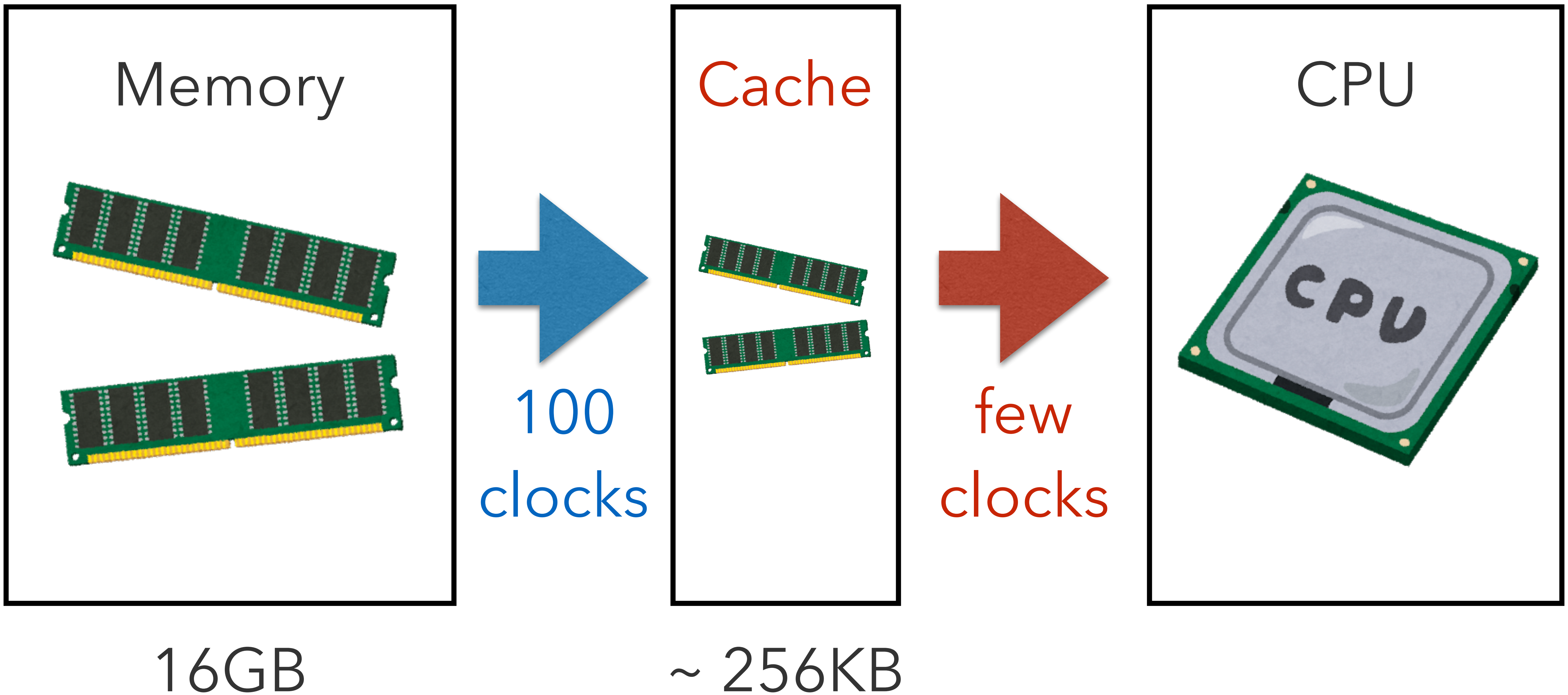
- Store data
- Clock frequency: ≒2133MHz
- Takes 100 operations to transmit data

(ref: https://www.agner.org/optimize/instruction_tables.pdf)

→ Memory is 100 times slower…

Computer Architecture (My computer)

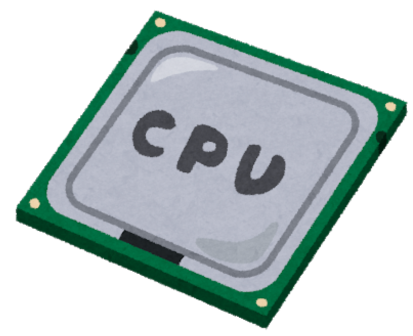
To make a full use of CPU power…

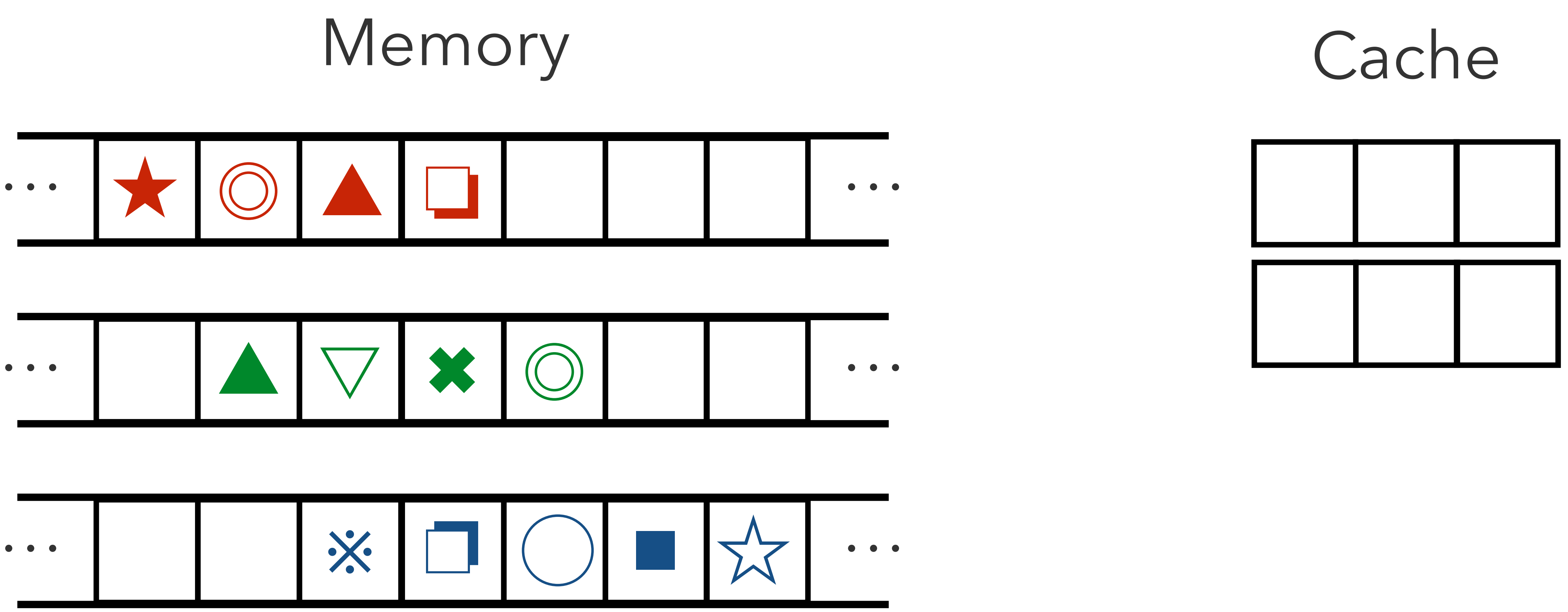


Cache Algorithm

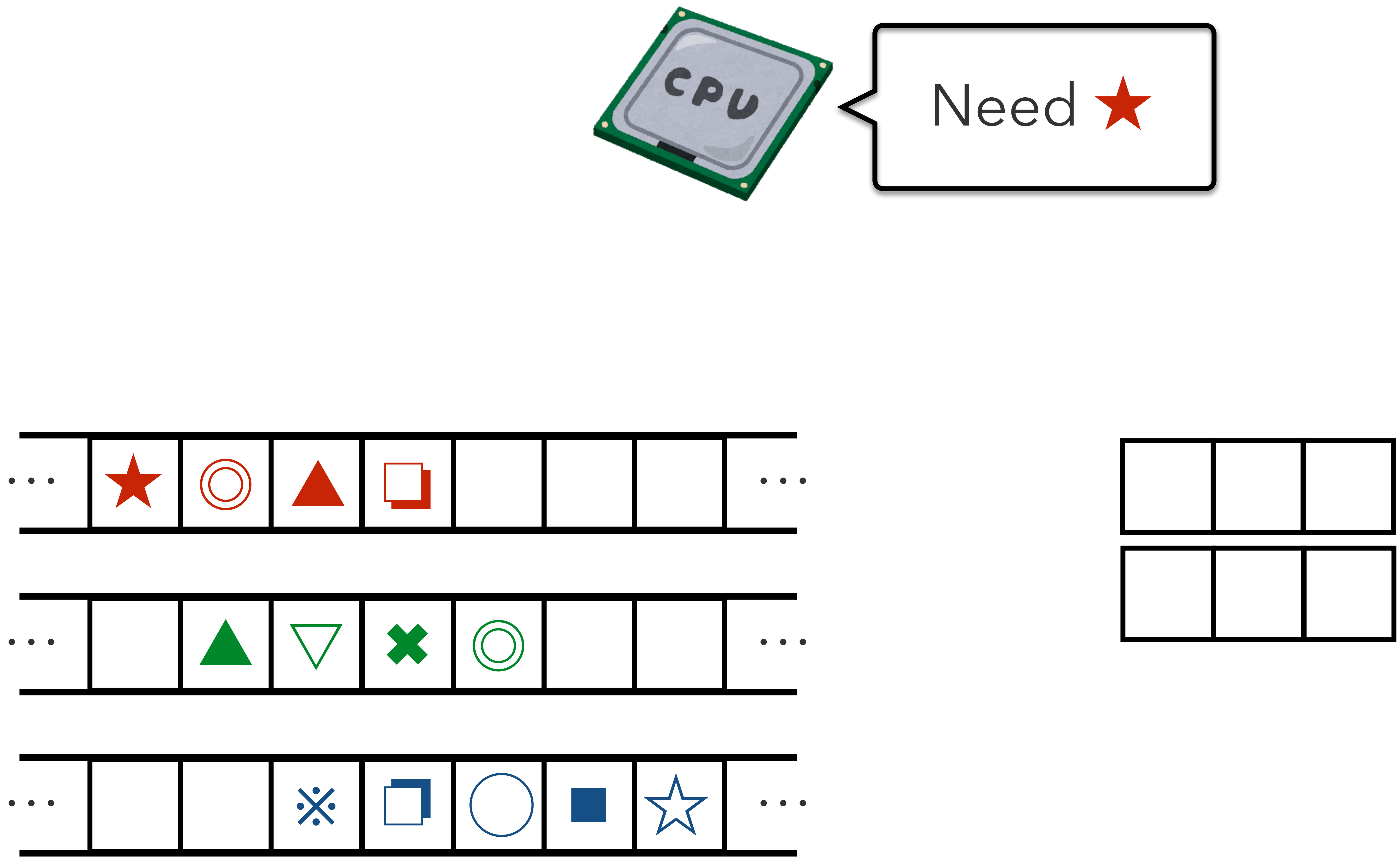
- LRU Algorithm
  1. Data is loaded from memory by chunk
  2. Don’t load a data if we have the data in cache
  3. If we use up the cache, delete the least recently used data

Cache Algorithm

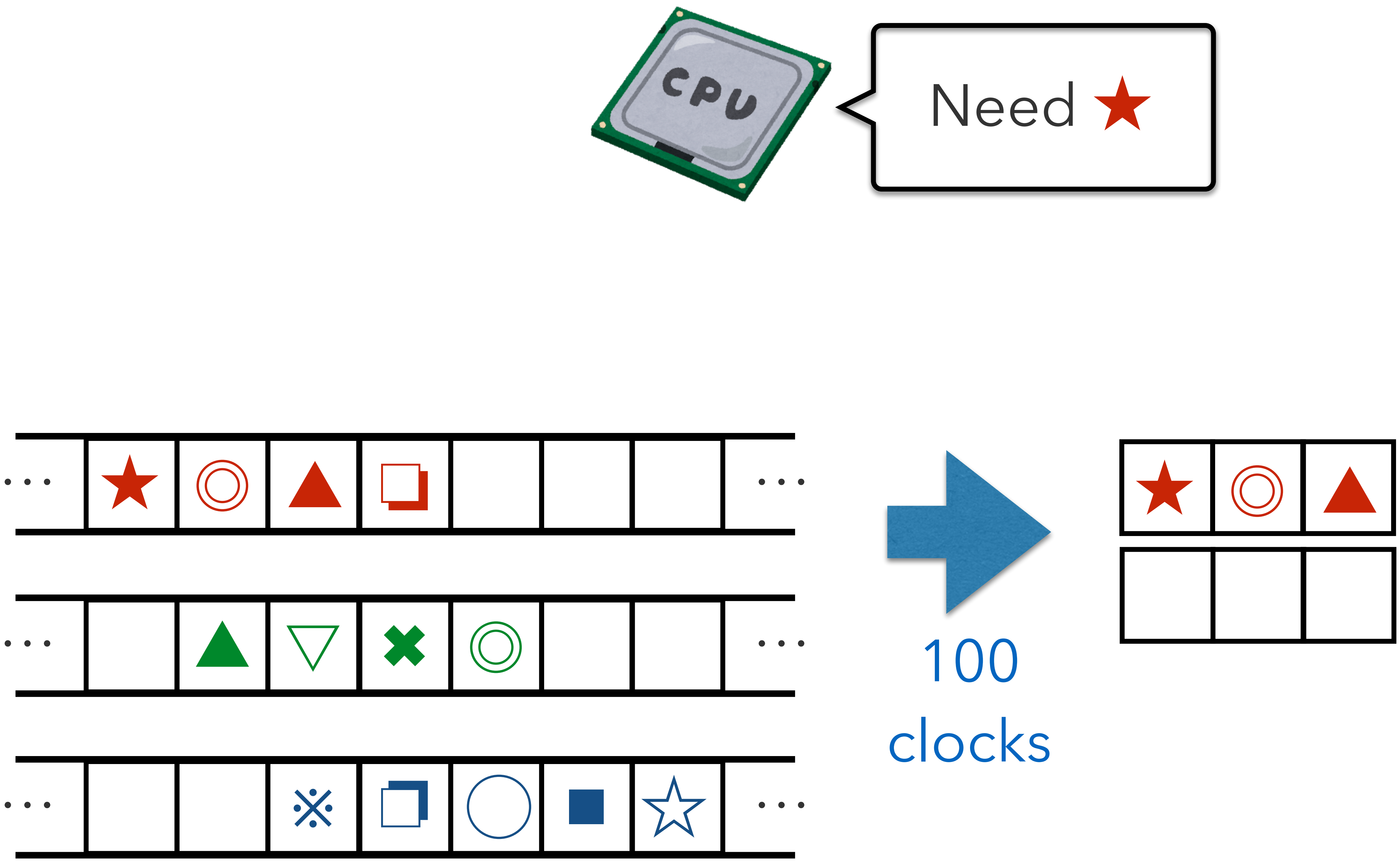




Cache Algorithm

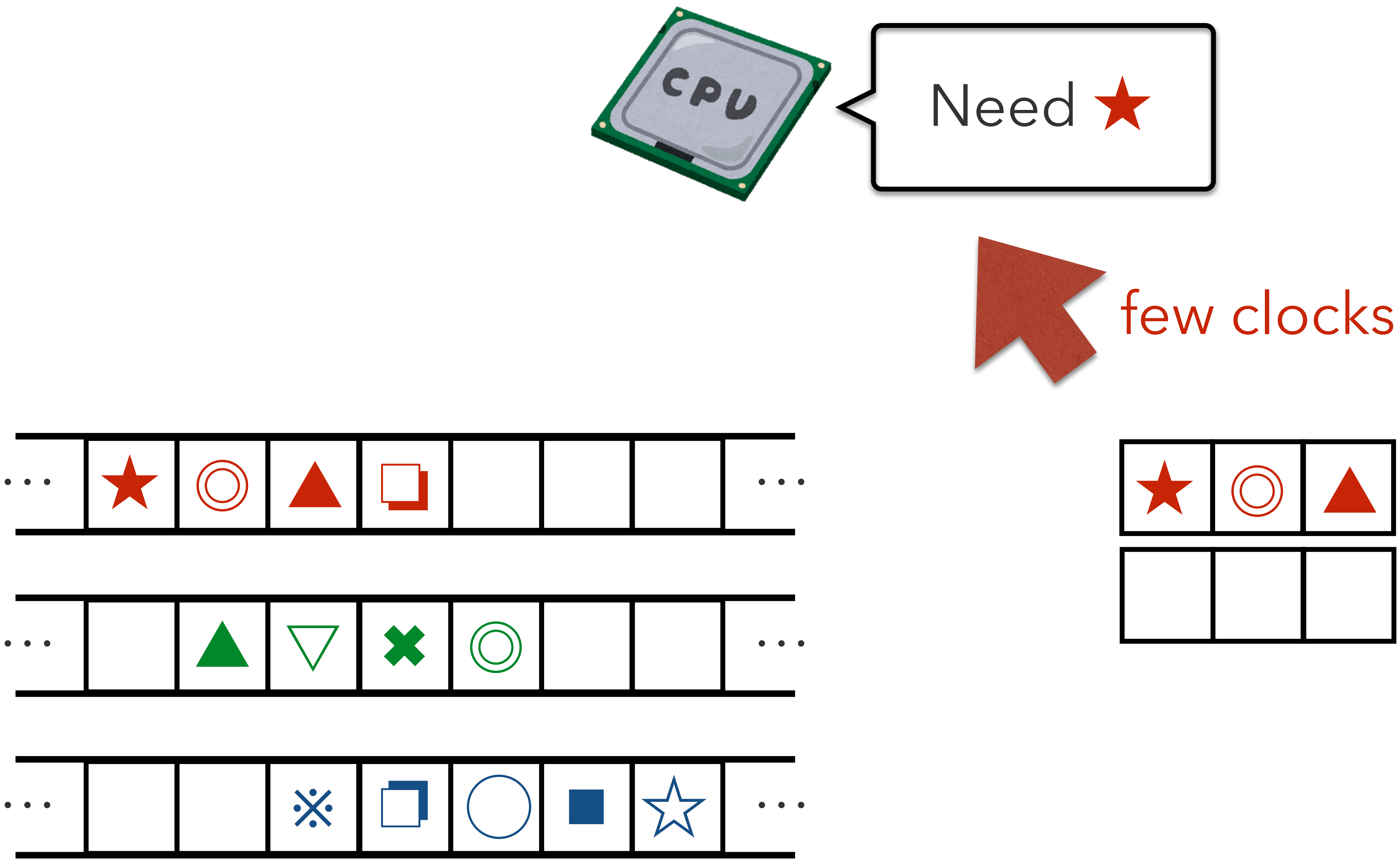


Cache Algorithm

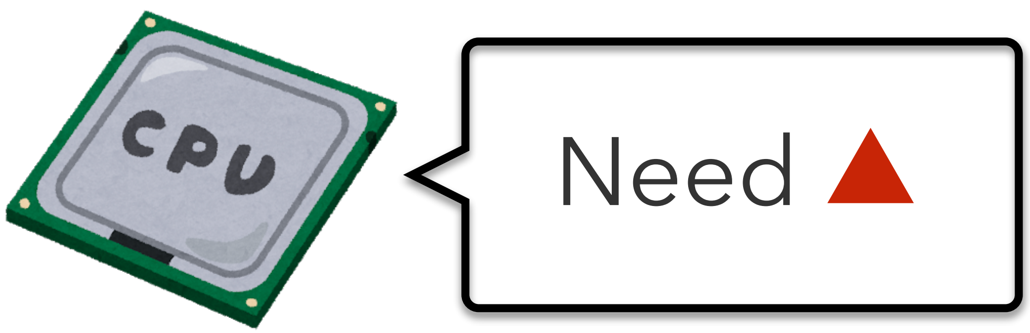


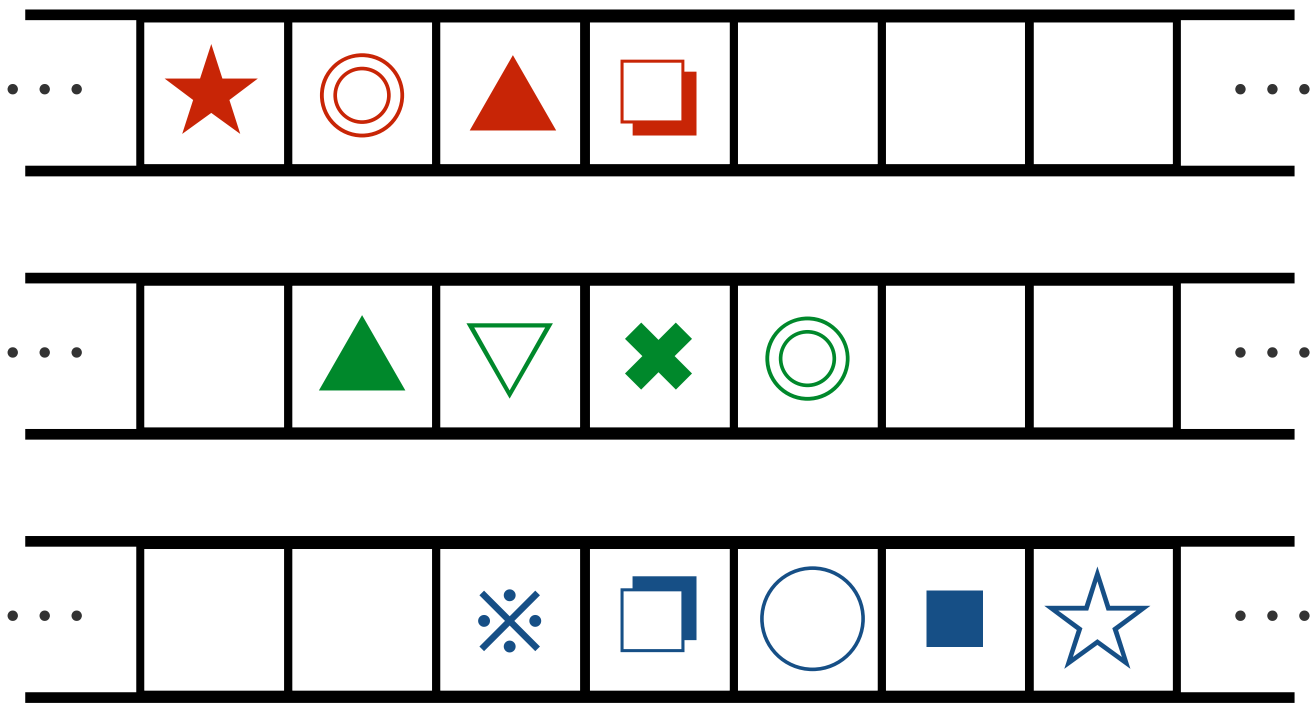
Data is loaded from memory by chunk

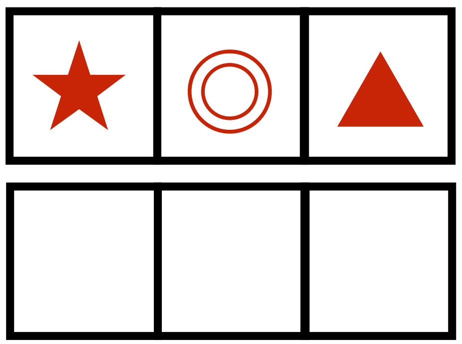
Cache Algorithm



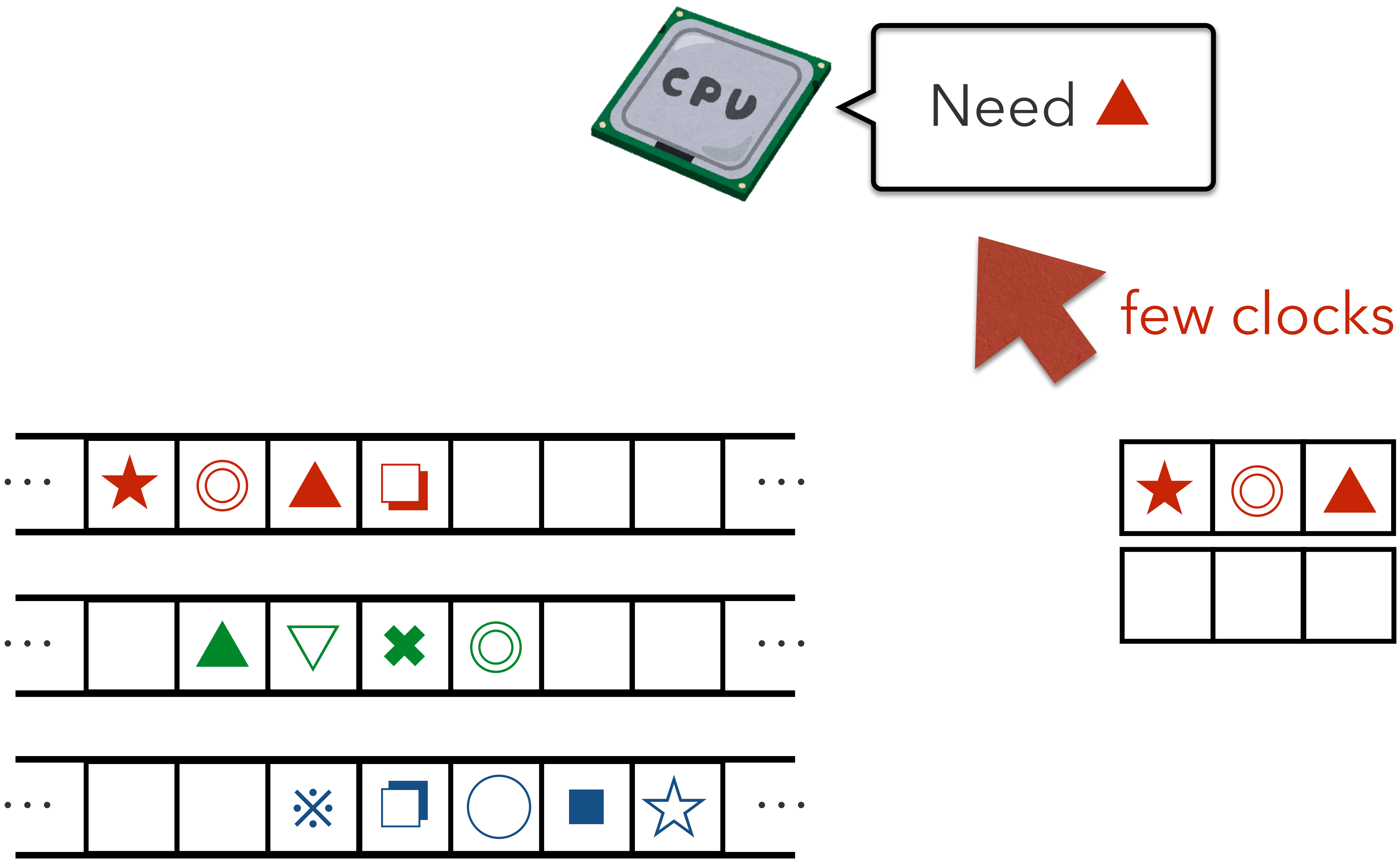
Cache Algorithm





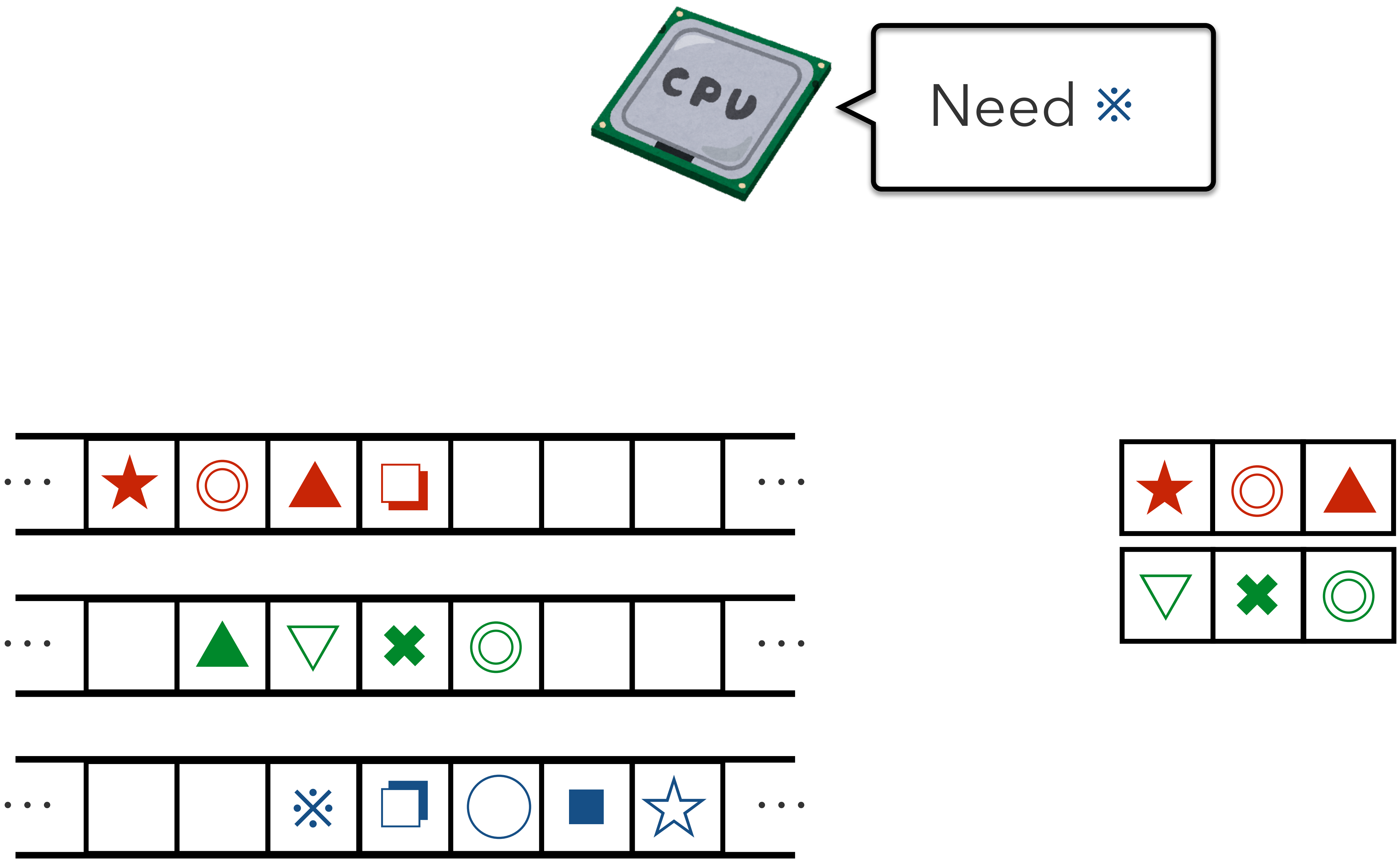


Cache Algorithm



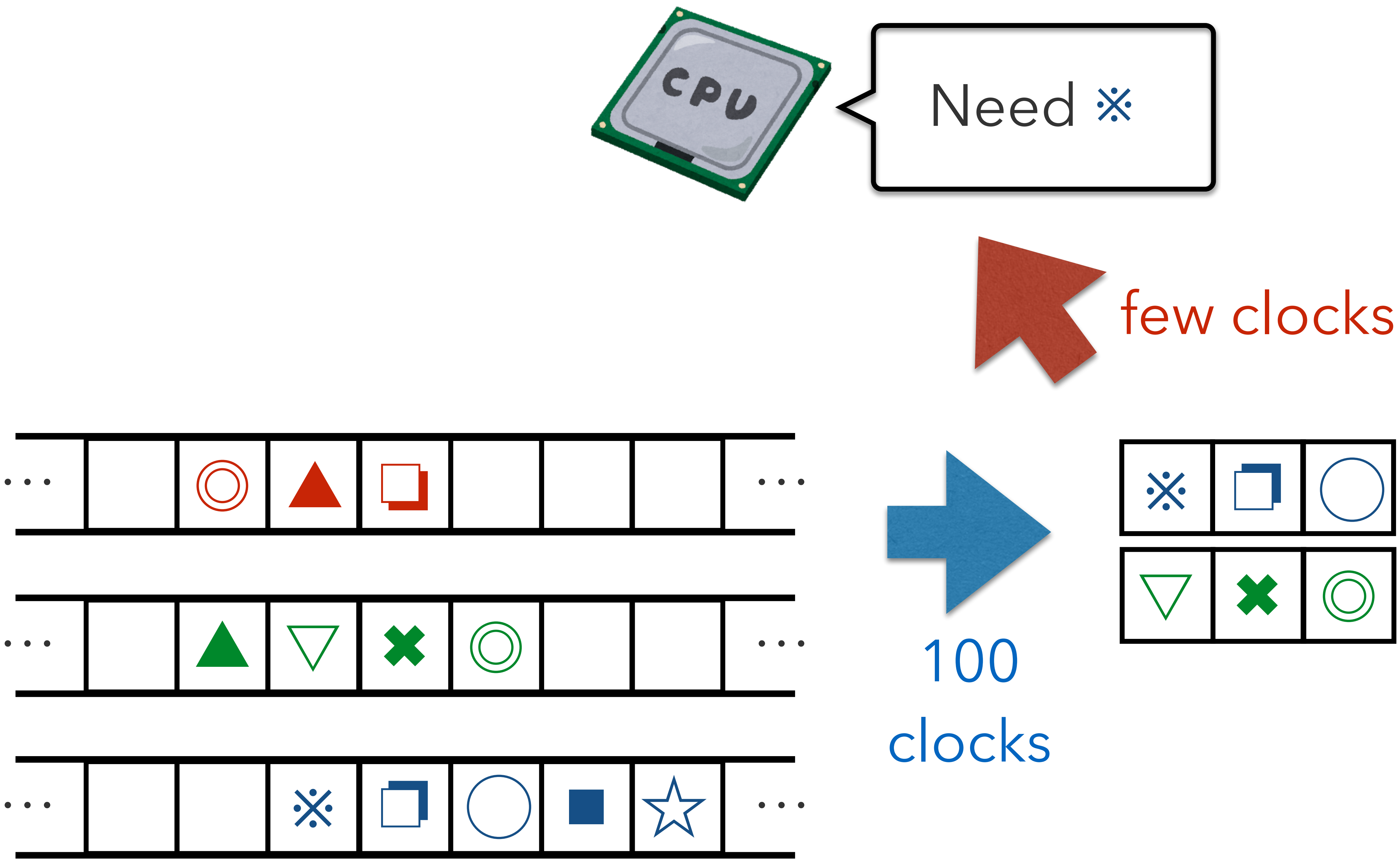
Don’t load data if we have it in cache

Cache Algorithm



If we use up the cache…

Cache Algorithm



delete the least recently used data (LRU Algorithm)

Two Codes

Code1

```
for (i = 0; i < MAT_SIZE; i++){
    for (j = 0; j < MAT_SIZE; j++){
        for (k = 0; k < MAT_SIZE; k++){
            c[i][j]+=a[i][k]*b[k][j];
        }
    }
}
```

128 [sec] (N=2000)

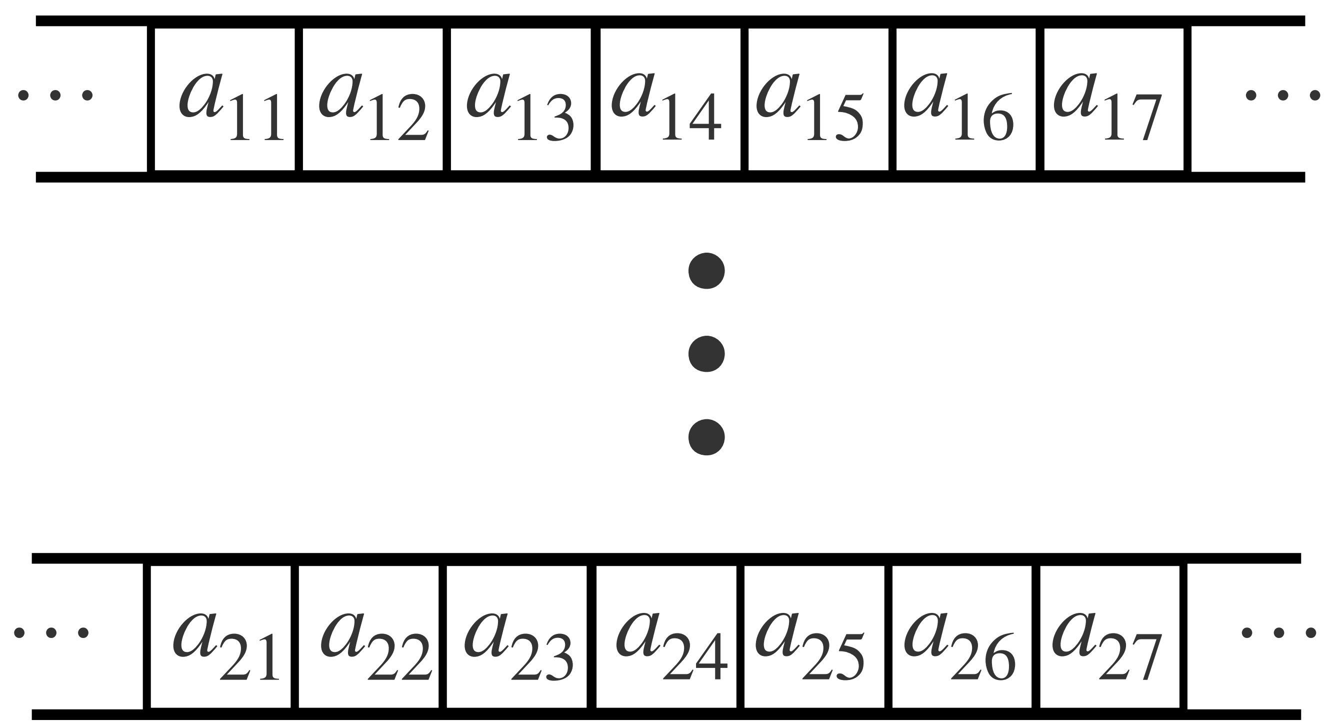
Code2

```
for (i = 0; i < MAT_SIZE; i++){
    for (k = 0; k < MAT_SIZE; k++){
        for (j = 0; j < MAT_SIZE; j++){
            c[i][j]+=a[i][k]*b[k][j];
        }
    }
}
```

22[sec] (N=2000)

Two-Dimensional Array in C

In C lang, a two-dimensional array is stored in row-wise



Cache in Code 1

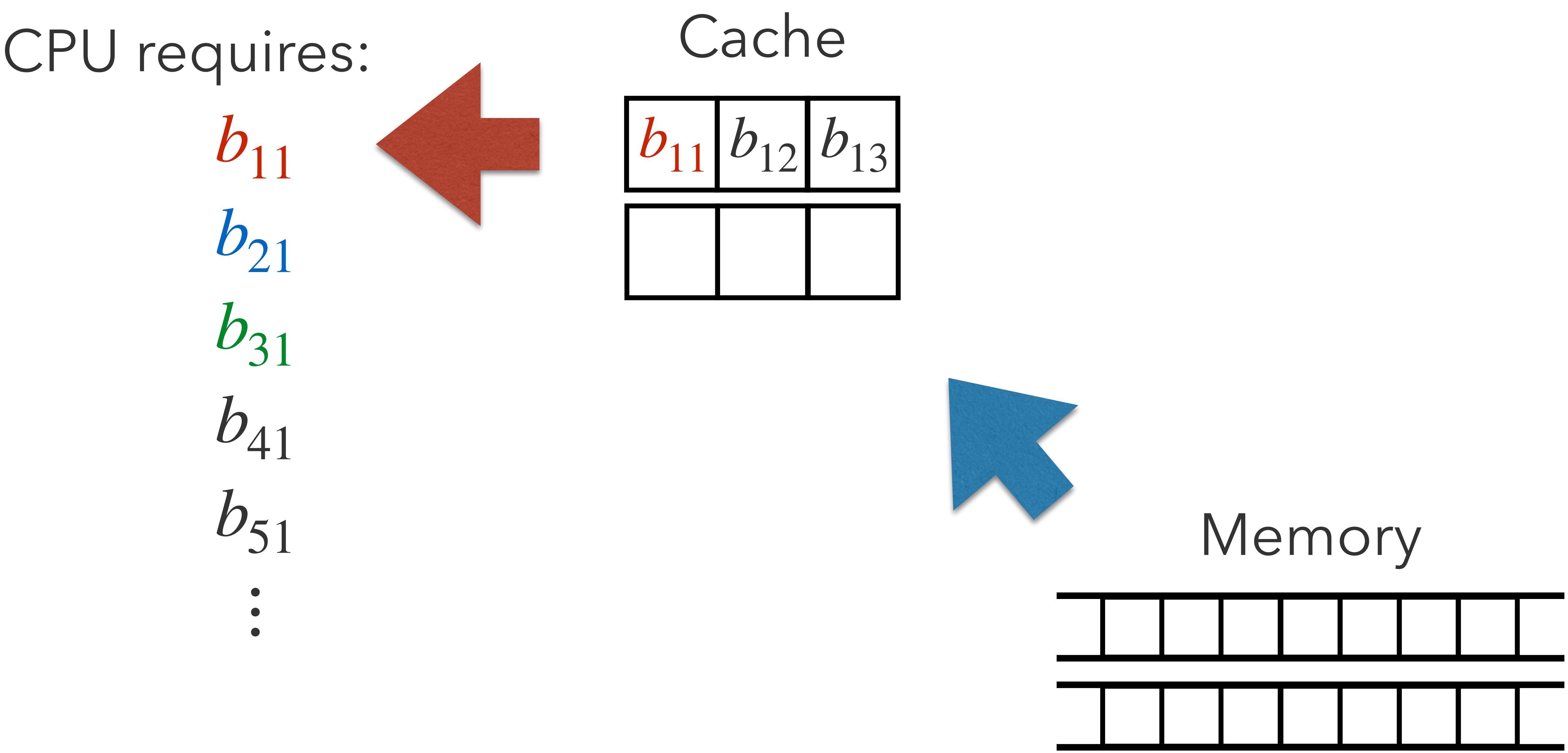
CPU requires:

$$b_{11}$$
$$b_{21}$$
$$b_{31}$$
$$b_{41}$$
$$b_{51}$$
$$\vdots$$

Code1

```
for (i = 0; i < MAT_SIZE; i++){
    for (j = 0; j < MAT_SIZE; j++){
        for (k = 0; k < MAT_SIZE; k++){
            c[i][j]+=a[i][k]*b[k][j];
        }
    }
}
```

Cache in Code 1



Cache in Code 1

CPU requires:

b_{11}

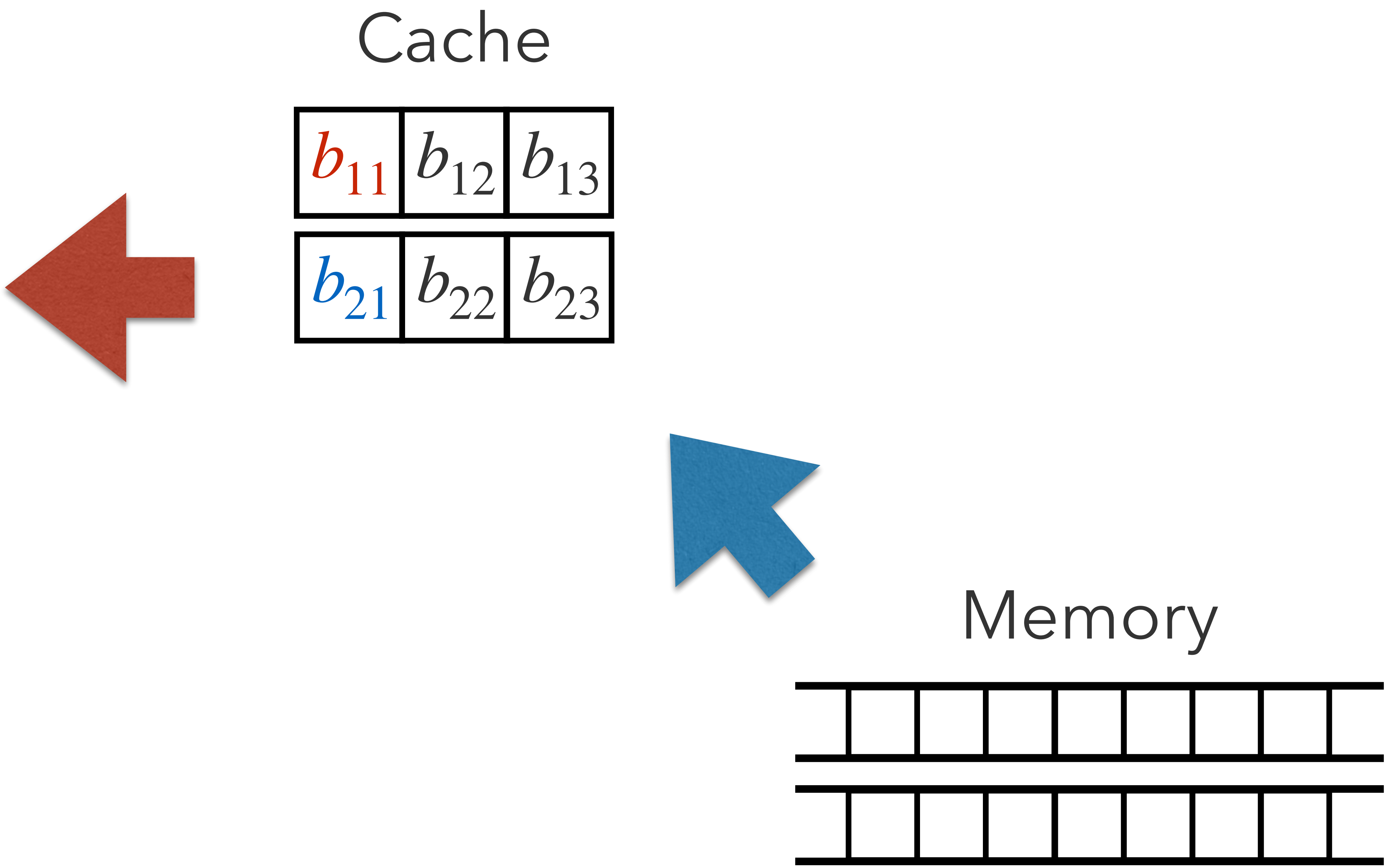
b_{21}

b_{31}

b_{41}

b_{51}

$\vdots$



Cache in Code 1

CPU requires:

b_{11}

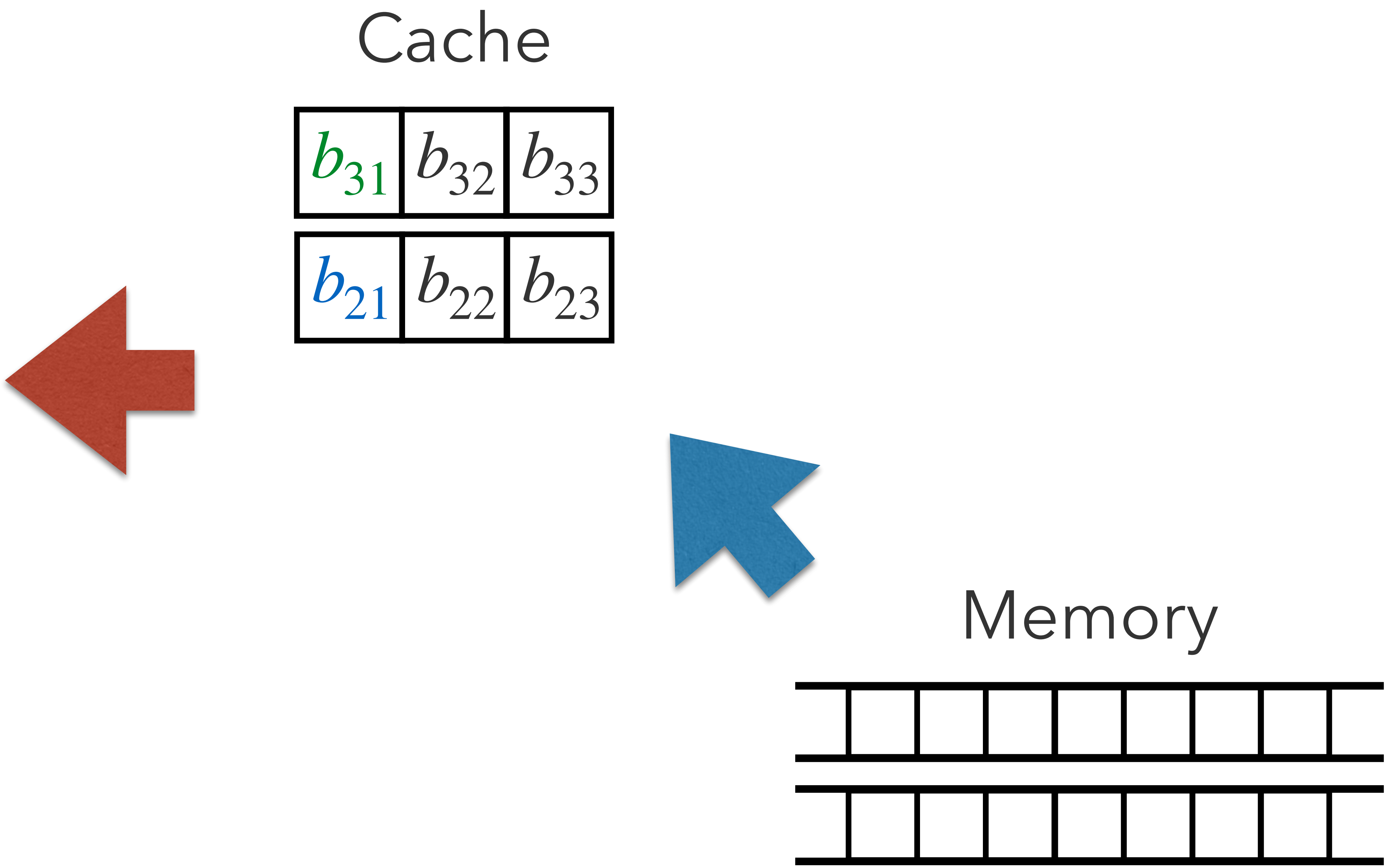
b_{21}

b_{31}

b_{41}

b_{51}

$\vdots$



Does not use cache at all !!

Cache in Code 2

CPU requires:

b_{11}

b_{12}

b_{13}

b_{14}

b_{15}

$\vdots$

Code2

```
for (i = 0; i < MAT_SIZE; i++){
    for (k = 0; k < MAT_SIZE; k++){
        for (j = 0; j < MAT_SIZE; j++){
            c[i][j]+=a[i][k]*b[k][j];
        }
    }
}
```

Cache in Code 2

CPU requires:

b_{11}

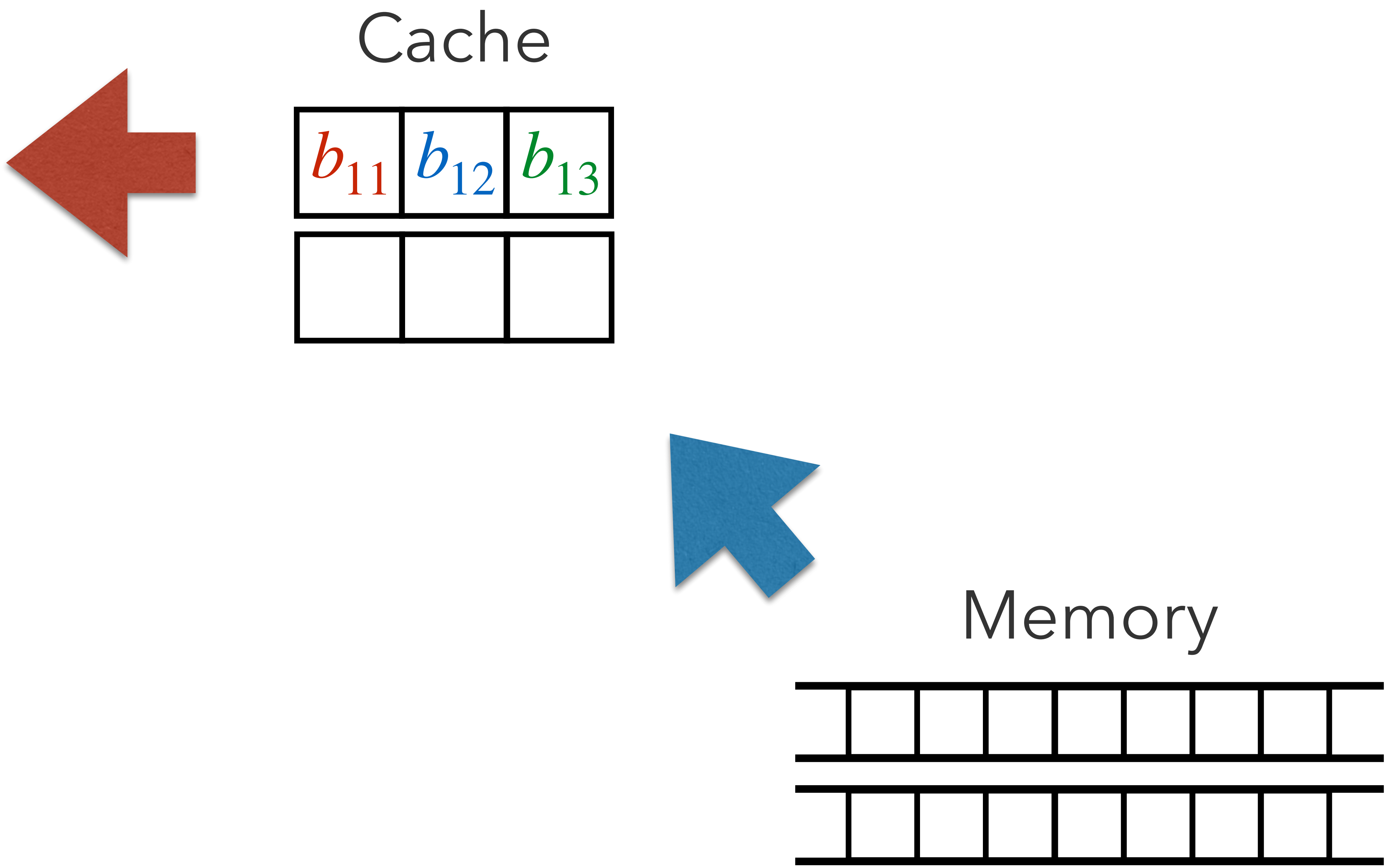
b_{12}

b_{13}

b_{14}

b_{15}

$\vdots$



Cache in Code 2

CPU requires:

b_{11}

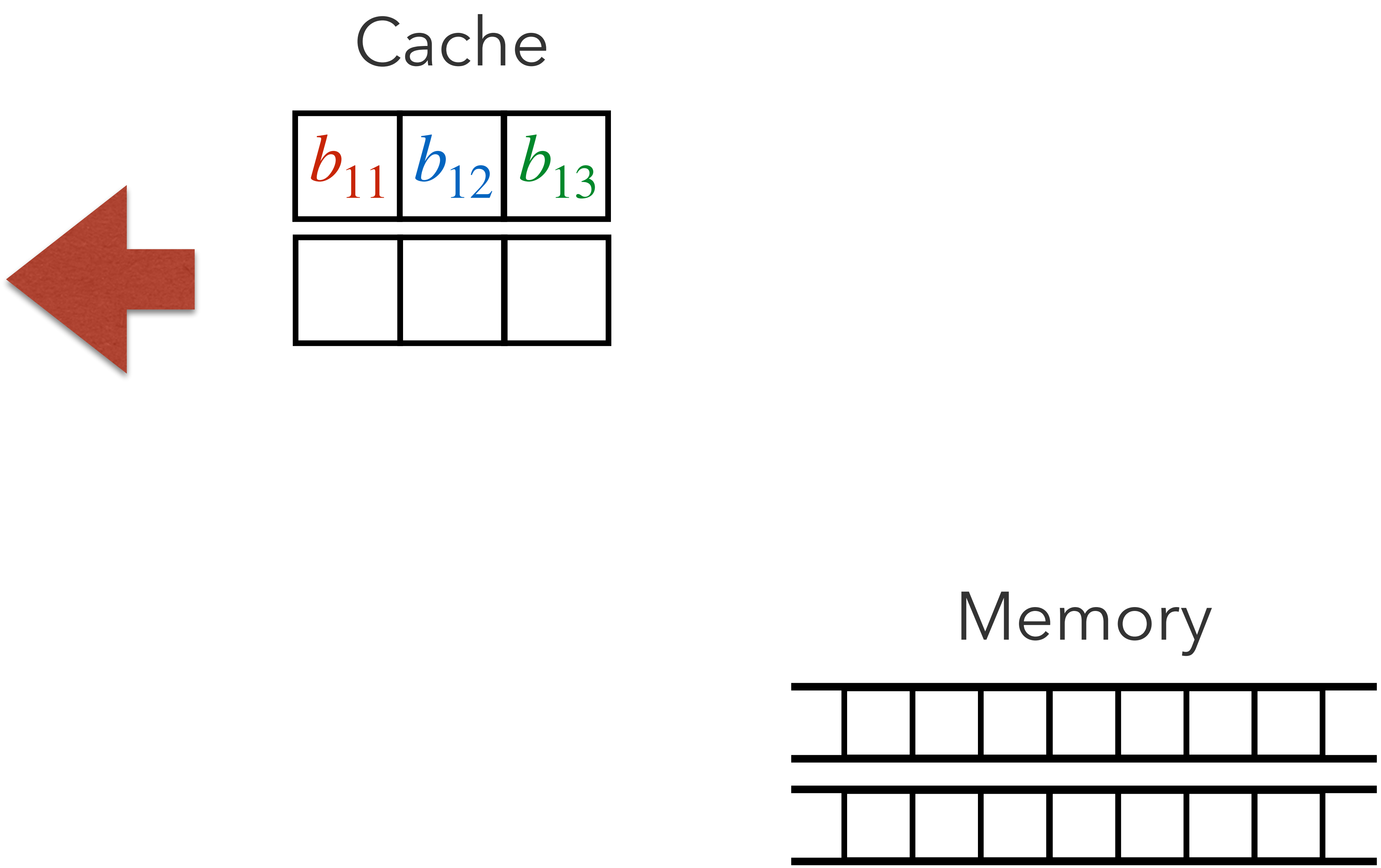
b_{12}

b_{13}

b_{14}

b_{15}

$\vdots$



Make full use of cache !

Summary

- Cache Hit
  - Number of times that CPU accesses data in cache
  - Key to high-performance code

Summary

Code1

```
for (i = 0; i < MAT_SIZE; i++){
    for (j = 0; j < MAT_SIZE; j++){
        for (k = 0; k < MAT_SIZE; k++){
            c[i][j]+=a[i][k]*b[k][j];
        }
    }
}
```

Low Cache Hit

Code2

```
for (i = 0; i < MAT_SIZE; i++){
    for (k = 0; k < MAT_SIZE; k++){
        for (j = 0; j < MAT_SIZE; j++){
            c[i][j]+=a[i][k]*b[k][j];
        }
    }
}
```

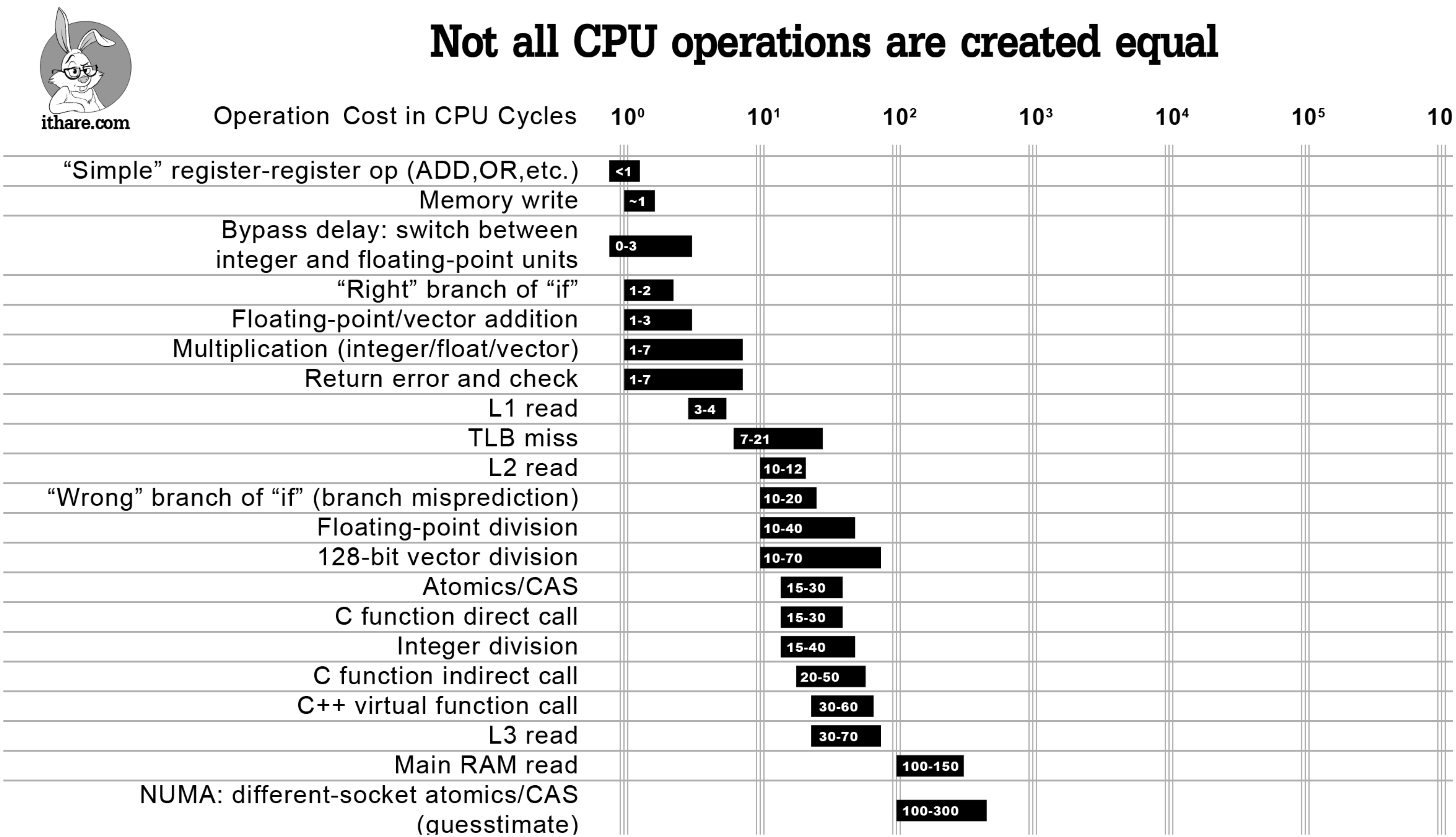
High Cache Hit

Loop Reorder:

= Change the order of loop to increase cache hit

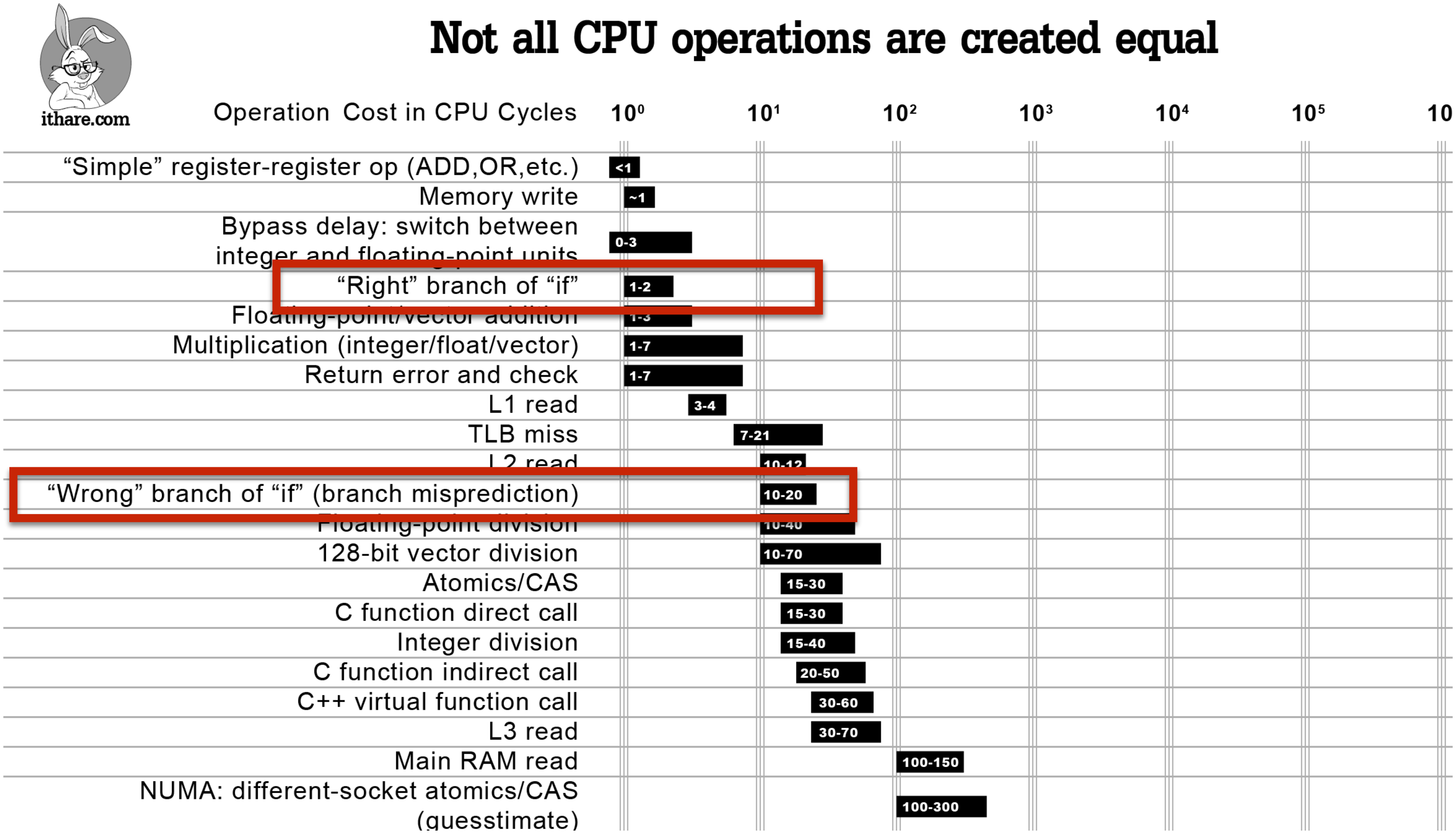
Other speed up technique

- CPU Bottlenecks



Other speed up technique

- CPU Bottlenecks



Other speed up technique

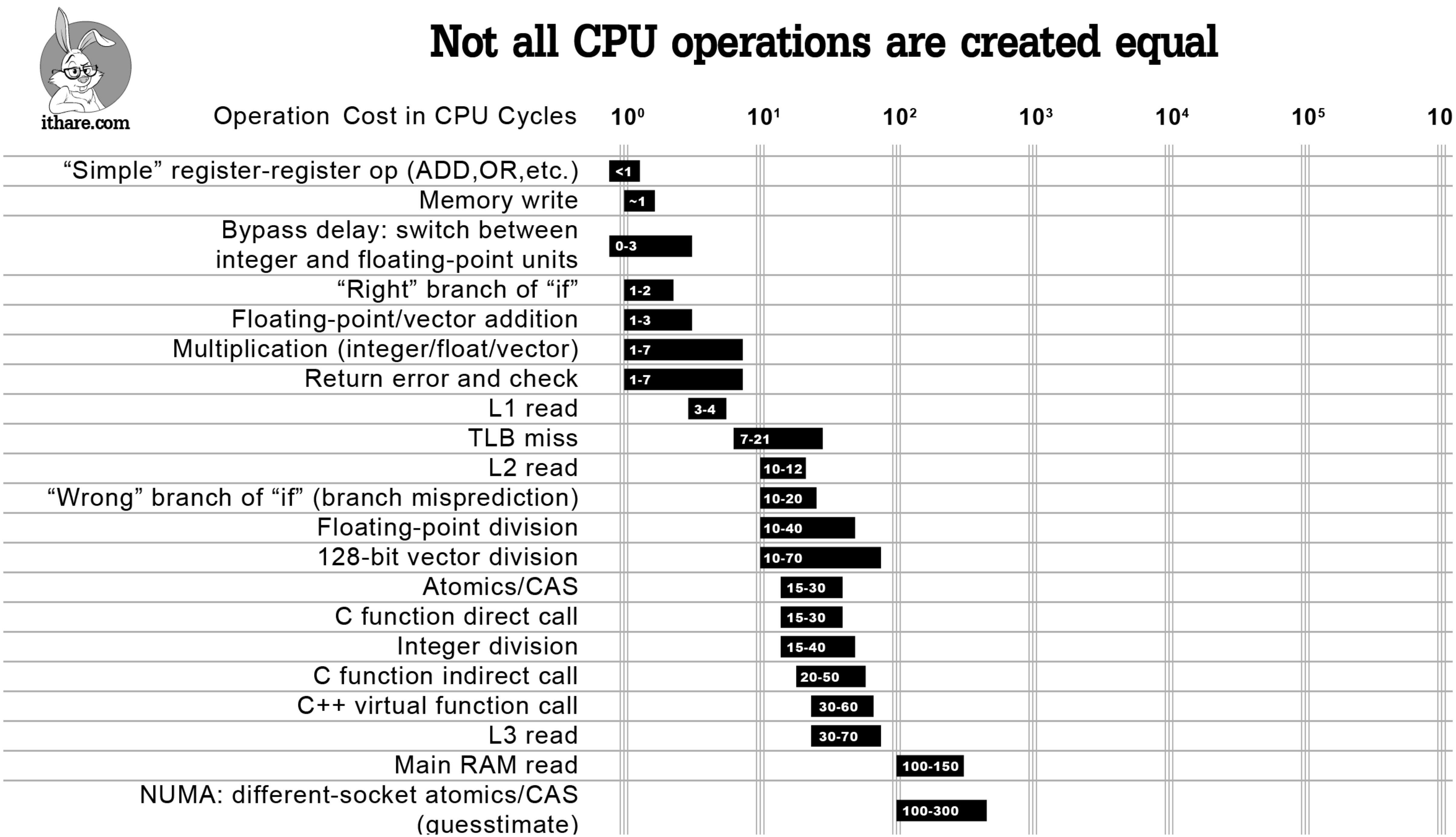
- Loop Unrolling

```
for (i = 0; i < MAT_SIZE; i+=4){
    i1 = i + 1; i2 = i + 2; i3 = i + 3;
    for (k = 0; k < MAT_SIZE; k+=4){
        k1 = k + 1; k2 = k + 2; k3 = k + 3;
        for (j = 0; j < MAT_SIZE; j++){
            c[i][j]+=a[i][k]*b[k][j];  c[i][j]+=a[i][k1]*b[k1][j];
            c[i][j]+=a[i][k2]*b[k2][j]; c[i][j]+=a[i][k3]*b[k3][j];

            c[i1][j]+=a[i1][k]*b[k][j];  c[i1][j]+=a[i1][k1]*b[k1][j];
            c[i1][j]+=a[i1][k2]*b[k2][j]; c[i1][j]+=a[i1][k3]*b[k3][j];

            c[i2][j]+=a[i2][k]*b[k][j];  c[i2][j]+=a[i2][k1]*b[k1][j];
            c[i2][j]+=a[i2][k2]*b[k2][j]; c[i2][j]+=a[i2][k3]*b[k3][j];

            c[i3][j]+=a[i3][k]*b[k][j];  c[i3][j]+=a[i3][k1]*b[k1][j];
            c[i3][j]+=a[i3][k2]*b[k2][j]; c[i3][j]+=a[i3][k3]*b[k3][j];
        }
    }
}
```

Speeds up few seconds…

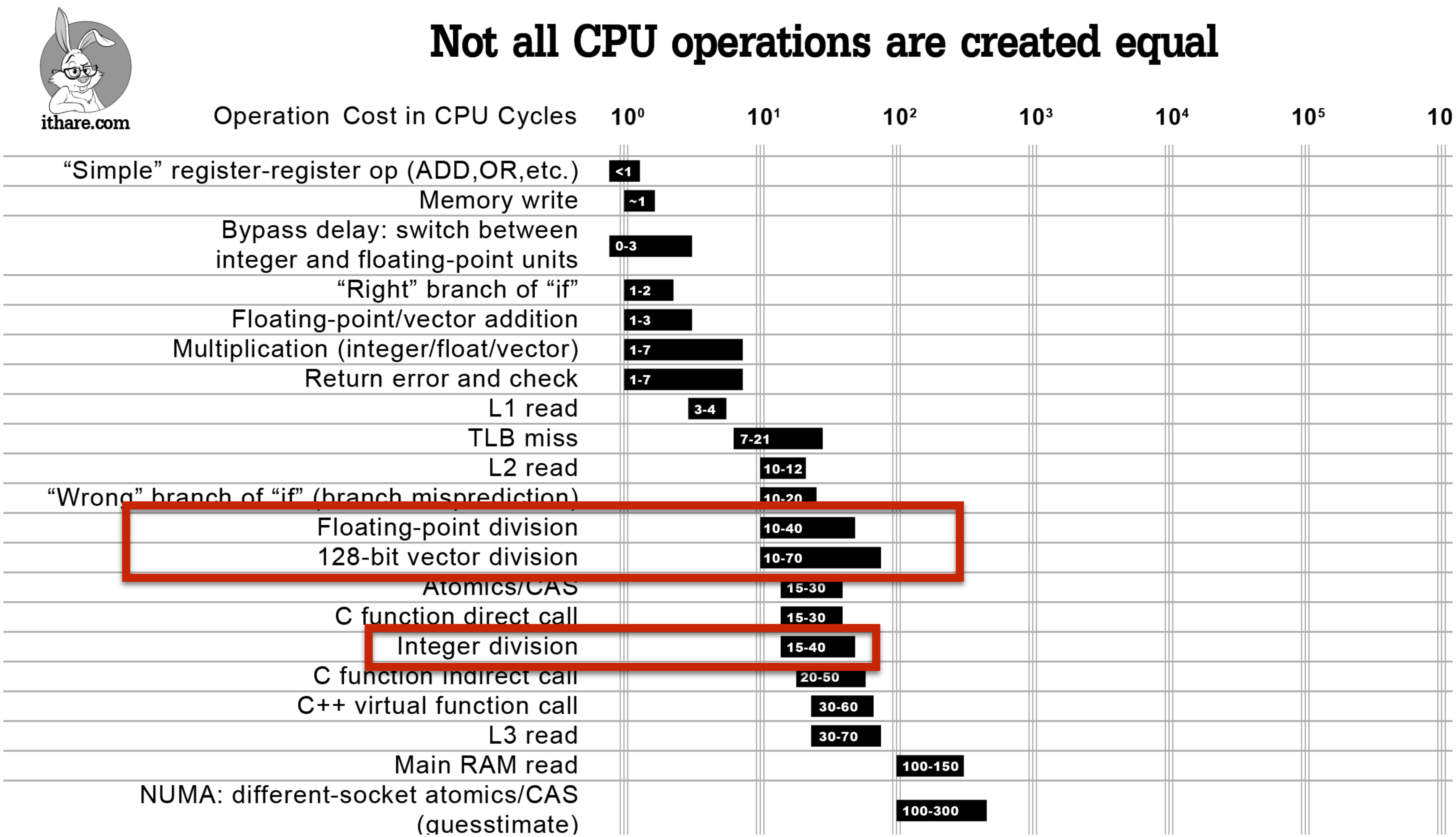
Other speed up technique

- CPU Bottlenecks



Other speed up technique

- CPU Bottlenecks



Other speed up technique

- Compute the sum of series: $S_N = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^N}$

Code1: $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^N}$

```
double res = 0.0;
double a = pow(2, SERIES_LEN);
for (i = 0; i < SERIES_LEN; i++){
    res += 1.0 / a;
    a /= 2.0;
}
```

14 [ms] (N=1000)

Code2: $\frac{1}{2^N}(1 + 2 + \cdots + 2^{N-1})$

```
double res = 0.0;
double a = 1.0;
for (i = 0; i < SERIES_LEN; i++){
    res += a;
    a *= 2.0;
}
res /= pow(2, SERIES_LEN);
```

7[ms] (N=1000)

Conclusion

- The bottleneck lies in

  Memory access / If-branch / Division

- High-Performance Programming tries to avoid them.

- Use Libraries! (BLAS etc.)

All codes are available at:
https://github.com/liyuan9988/TeaTalk-210610-Fast-Code