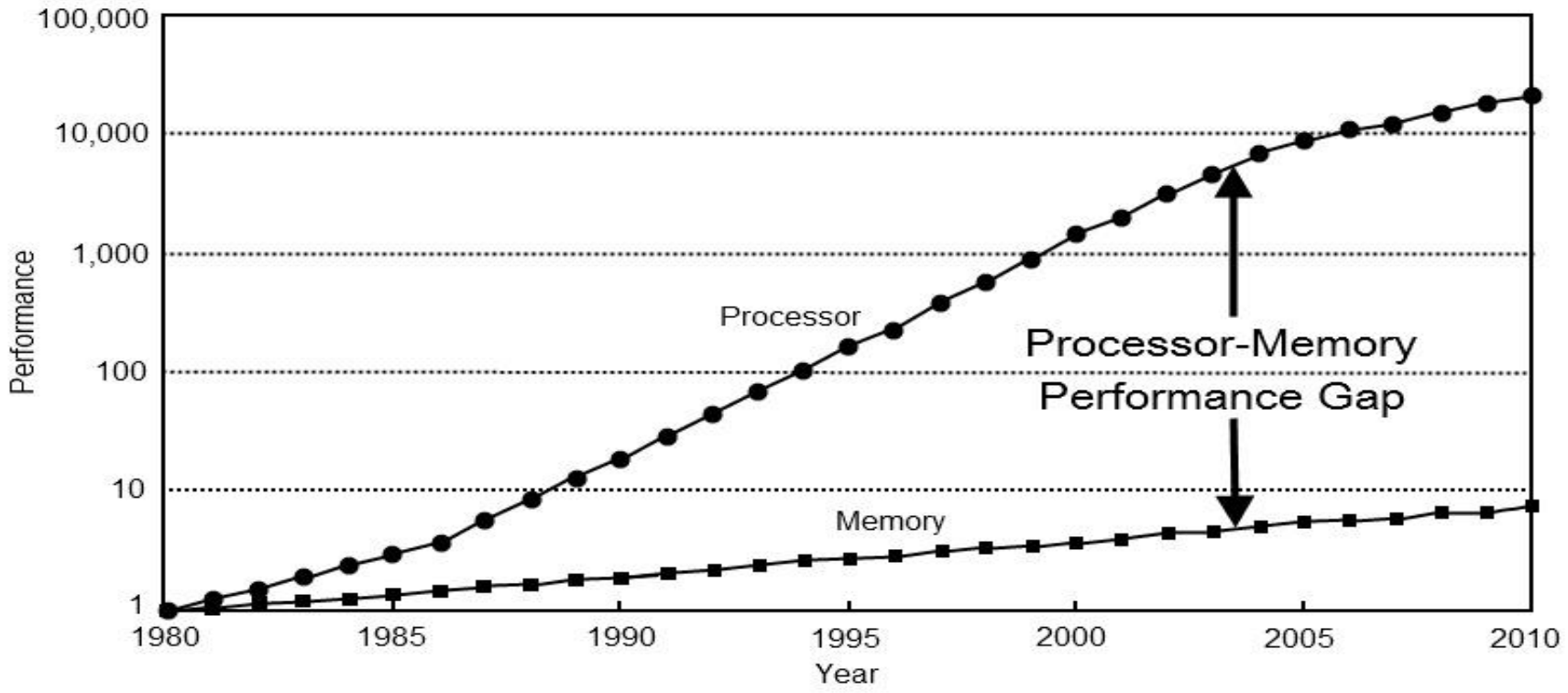
What Every Programmer Should Know About Memory

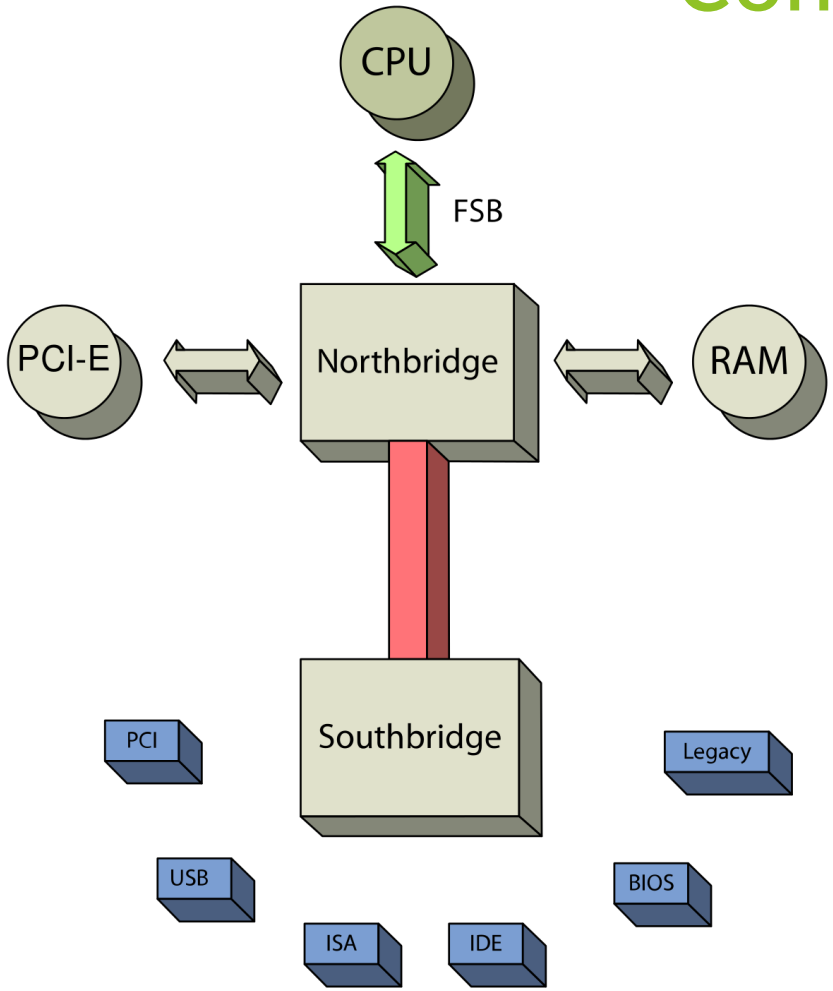
Borsik Gábor

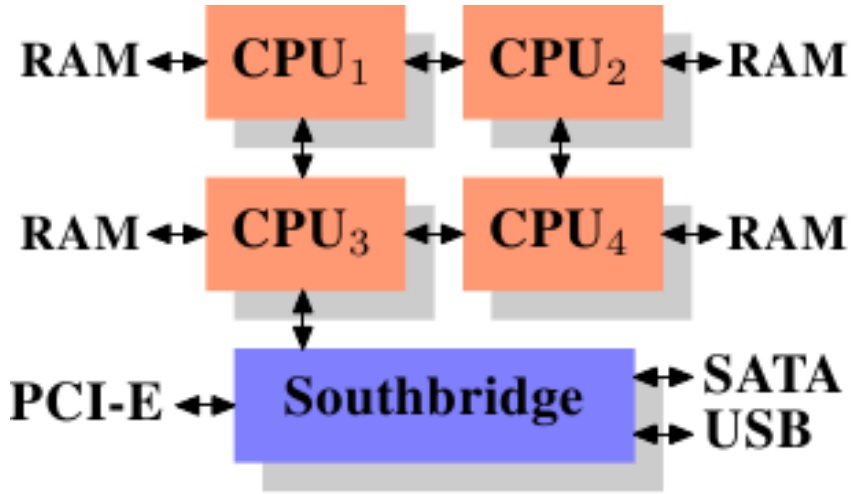
2018. 03. 21.

Why?


100,000
10,000
1,000
100
10
1
Performance
Processor
Memory
Processor-Memory
Performance Gap
1980
1985
1990
1995
2000
2005
2010
Year

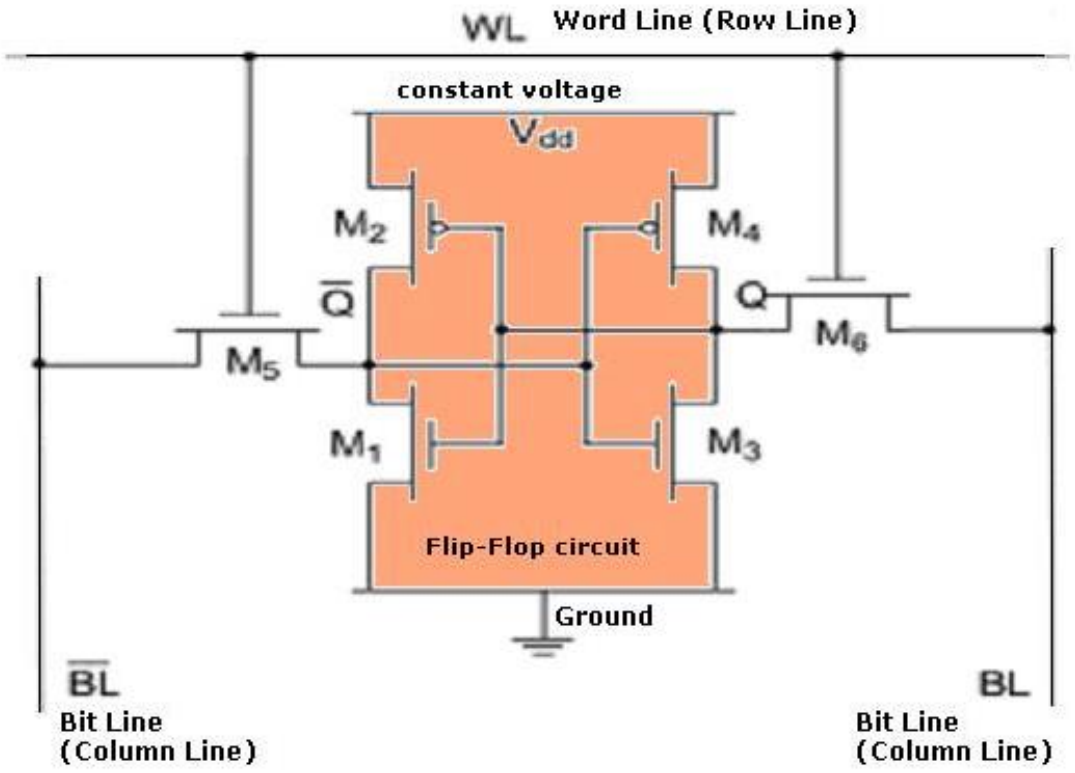
Commodity hardware





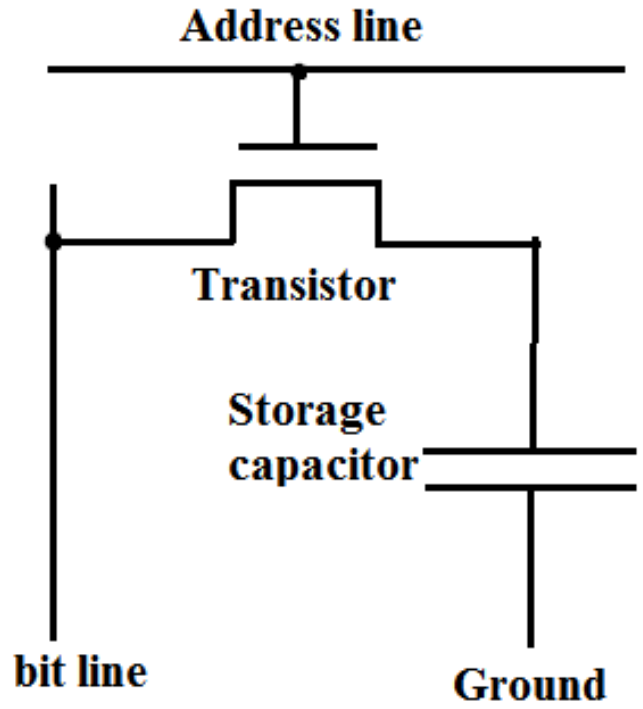
RAM types

Static random-access memory



- 6 transistors
- Two stable state
- Constant power

Dynamic random-access memory



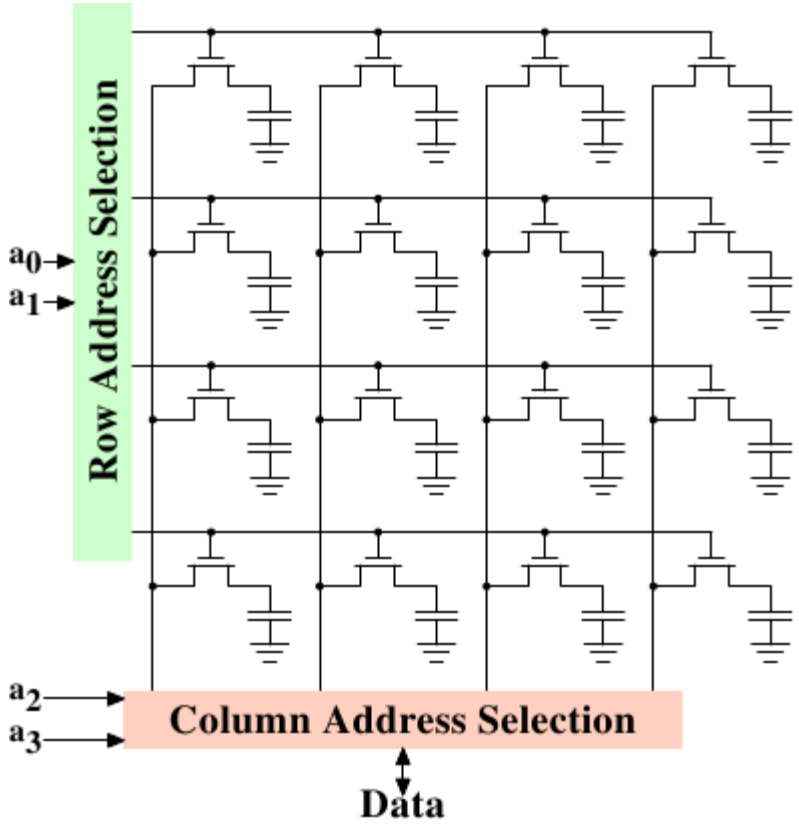
- 1 transistor, 1 capacitor
- Need refresh
- Signal problems

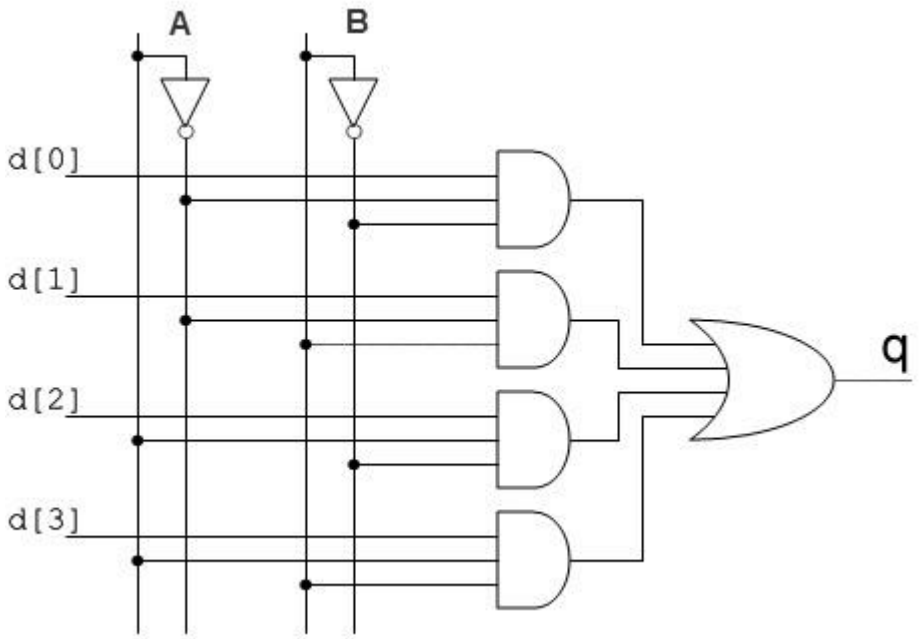
SRAM vs DRAM

- Speed
- Size
- Power consumption
- Price

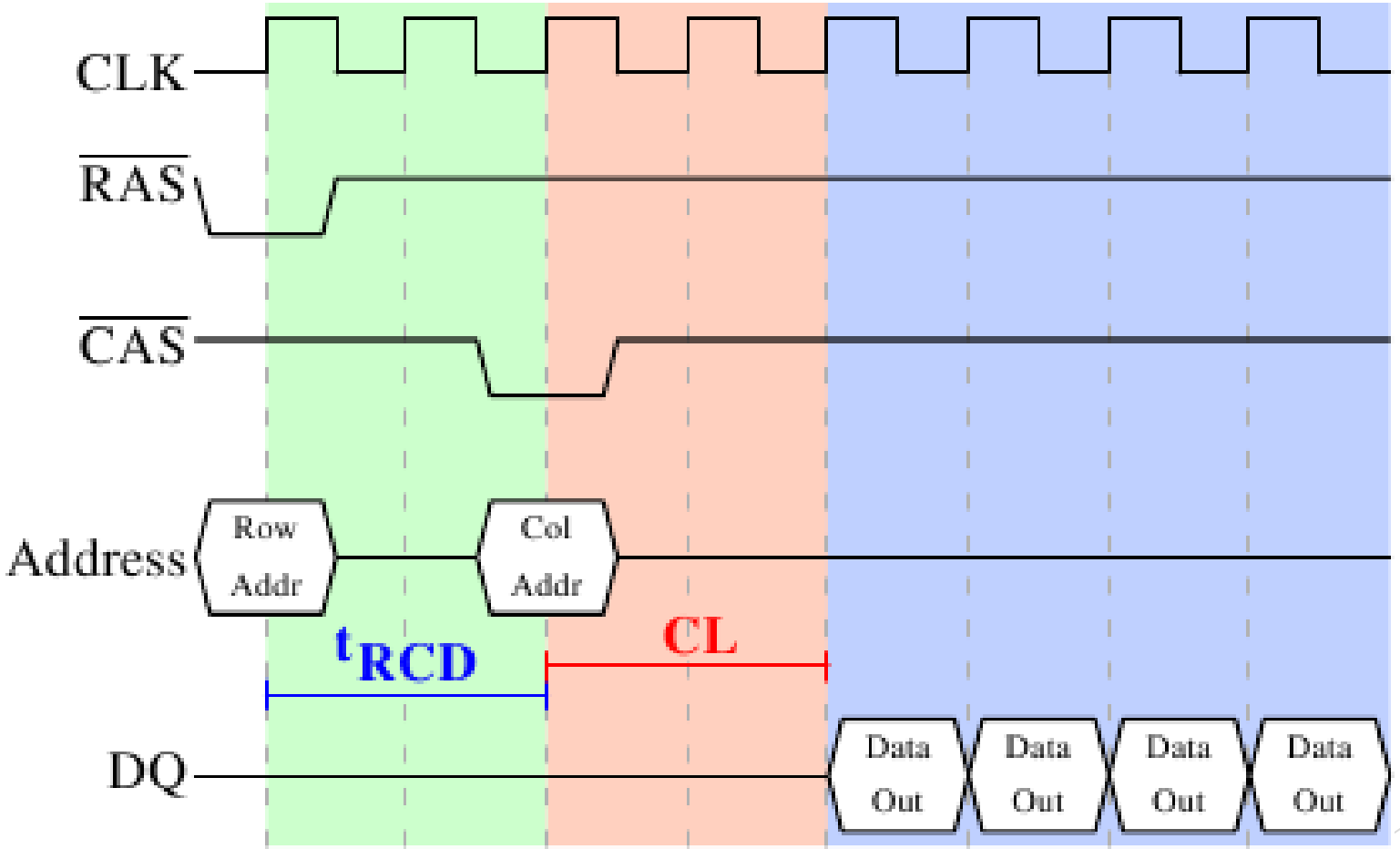
DRAM access

Dynamic RAM schematic

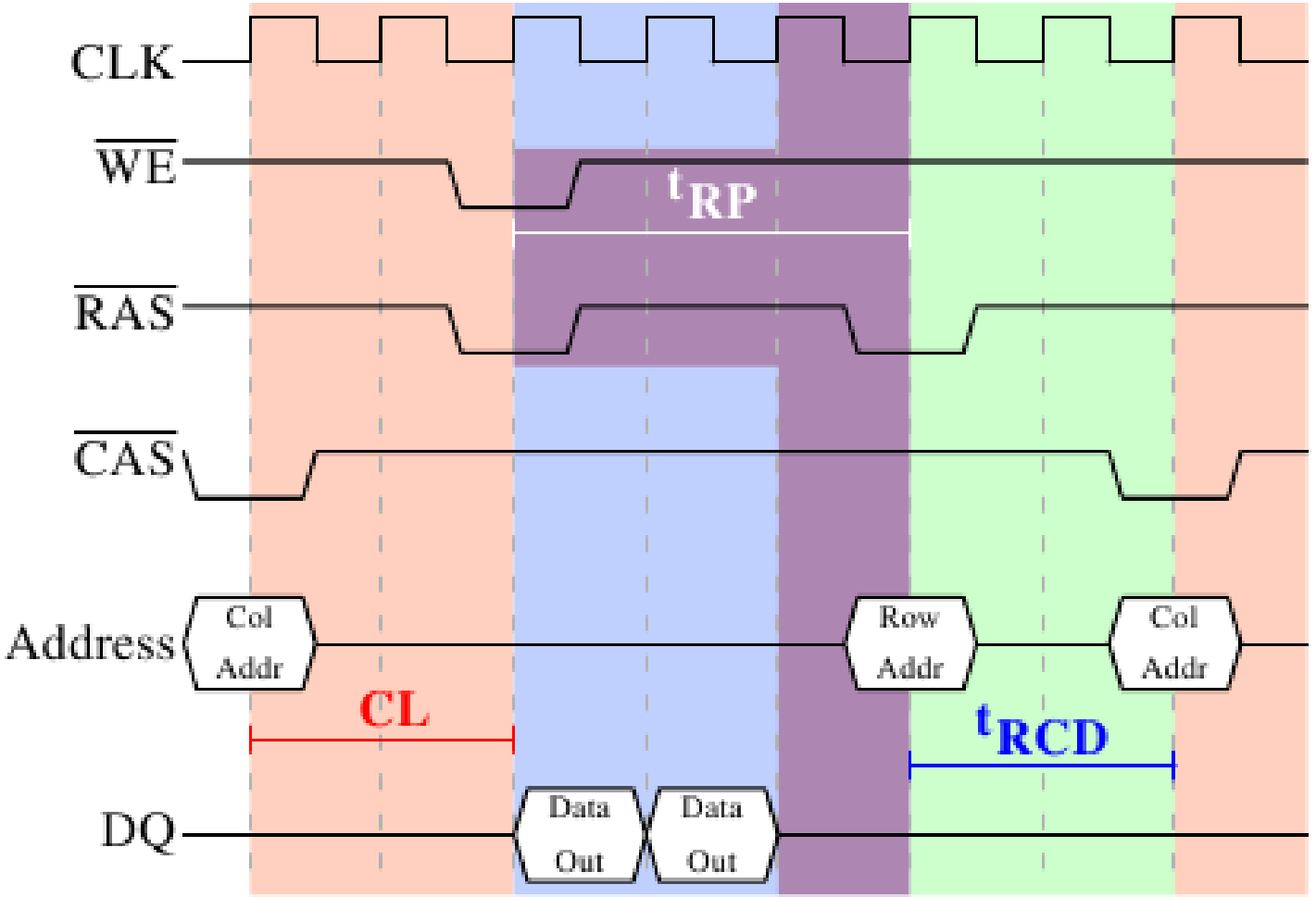




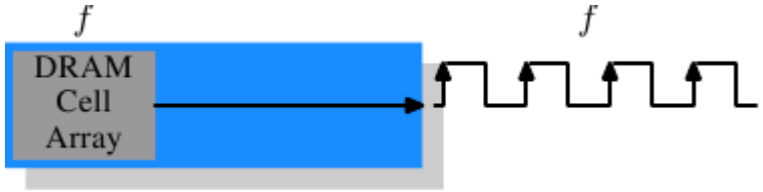
Synchronous DRAM



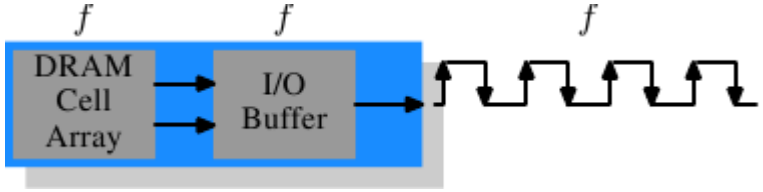
Synchronous DRAM



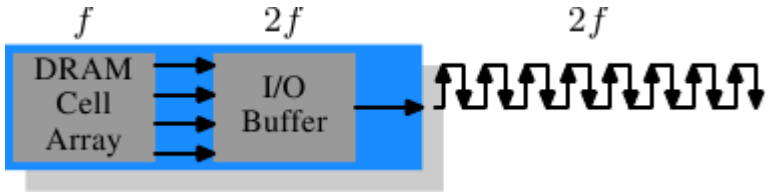
SDRAM types



SDR SDRAM



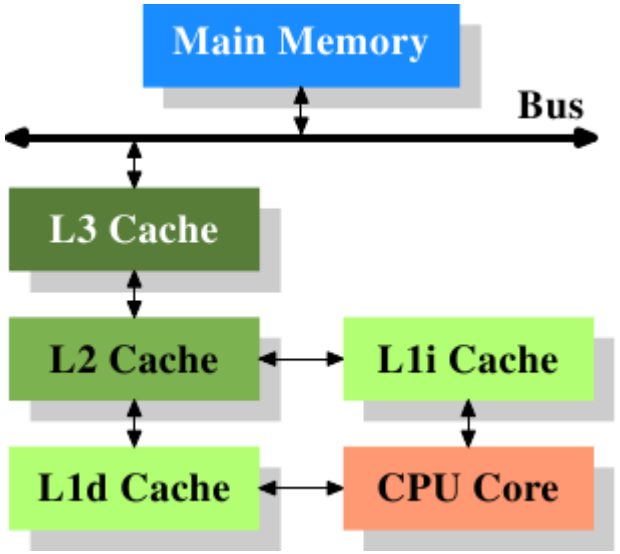
DDR1 SDRAM

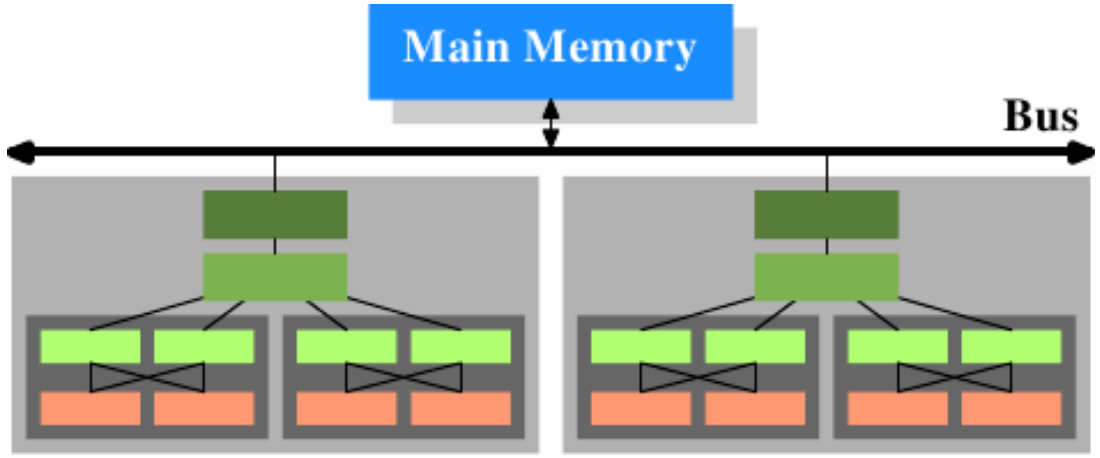


DDR2 SDRAM

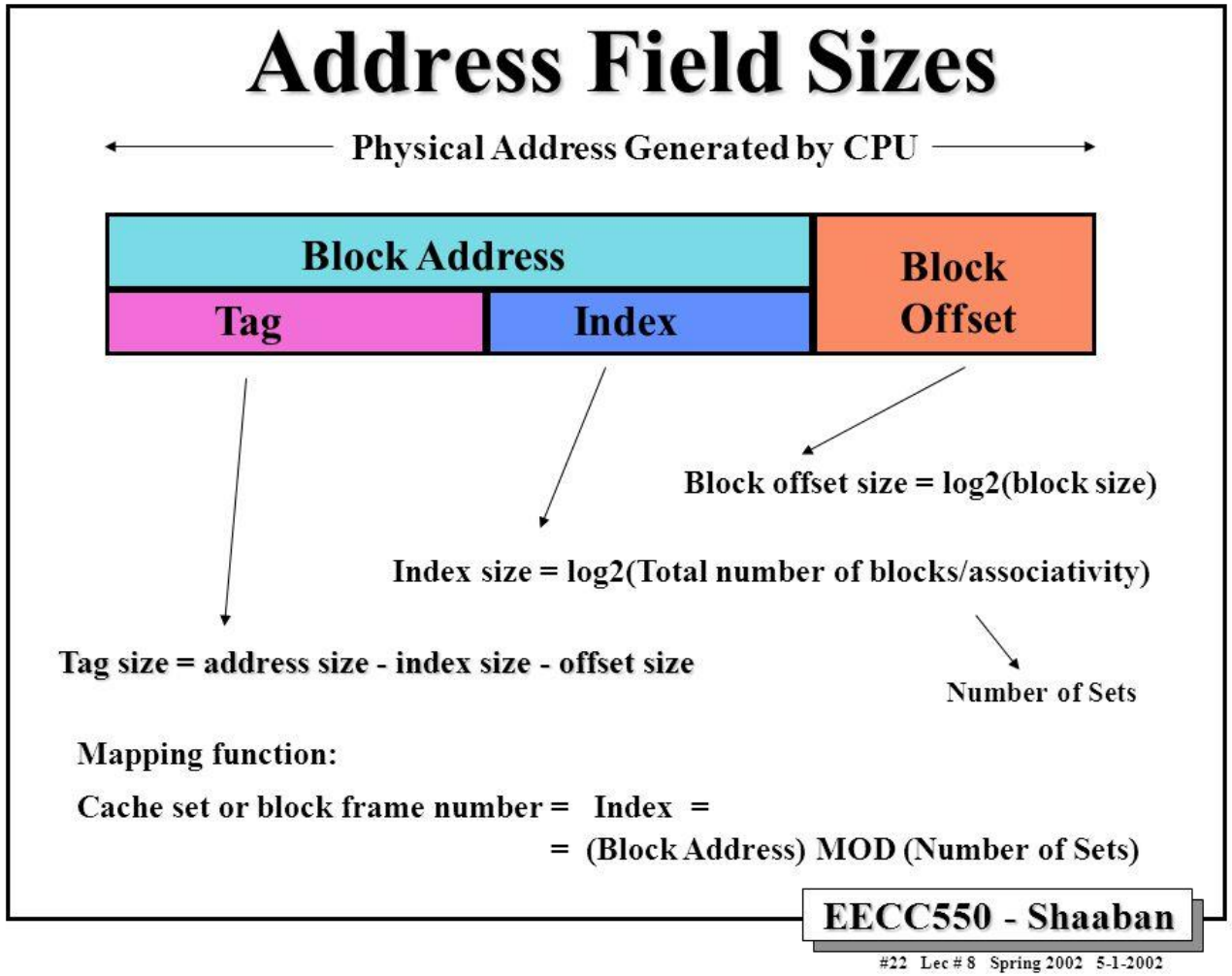
CPU Caches

CPU caches in the big picture

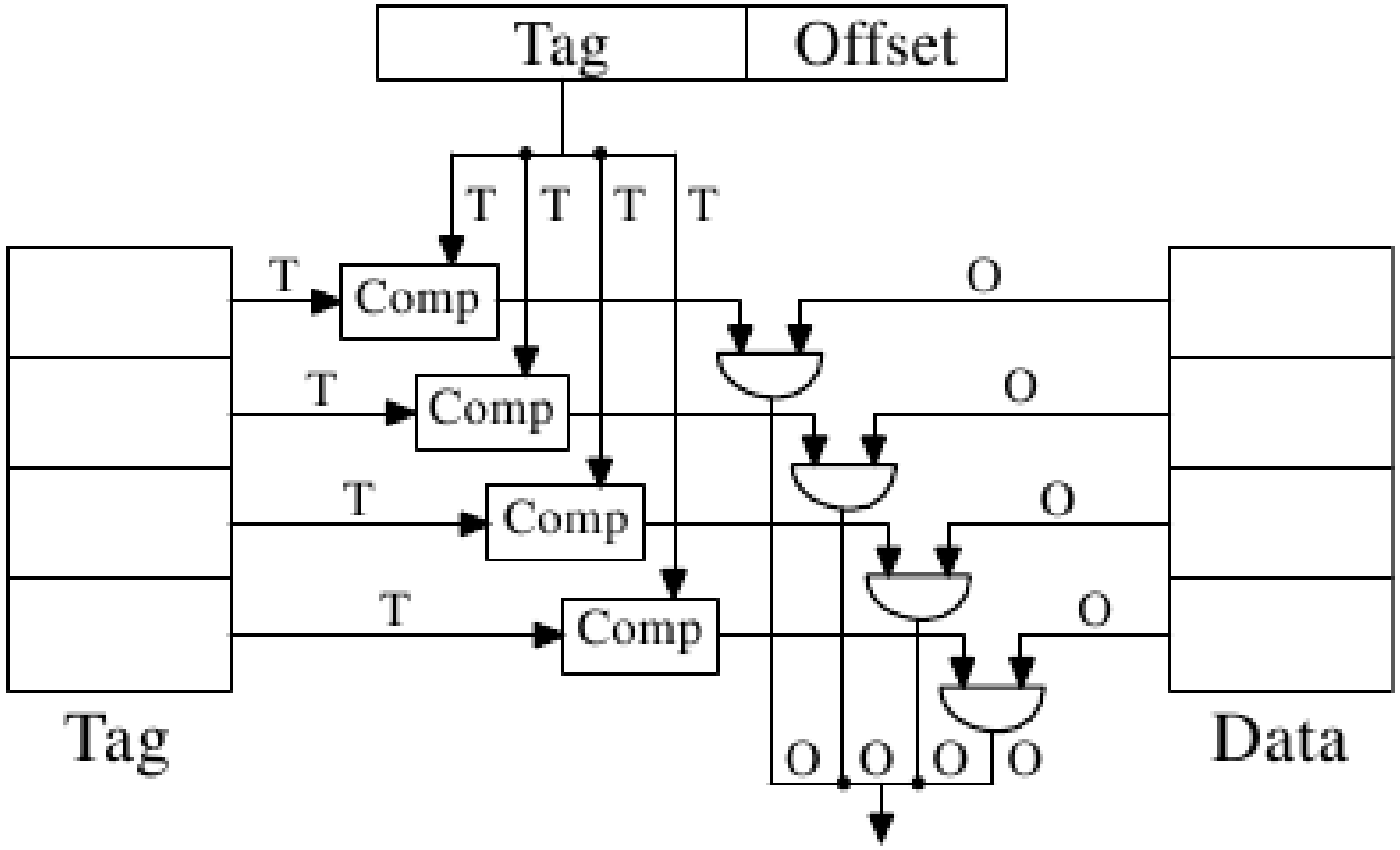




Cache operation at high level

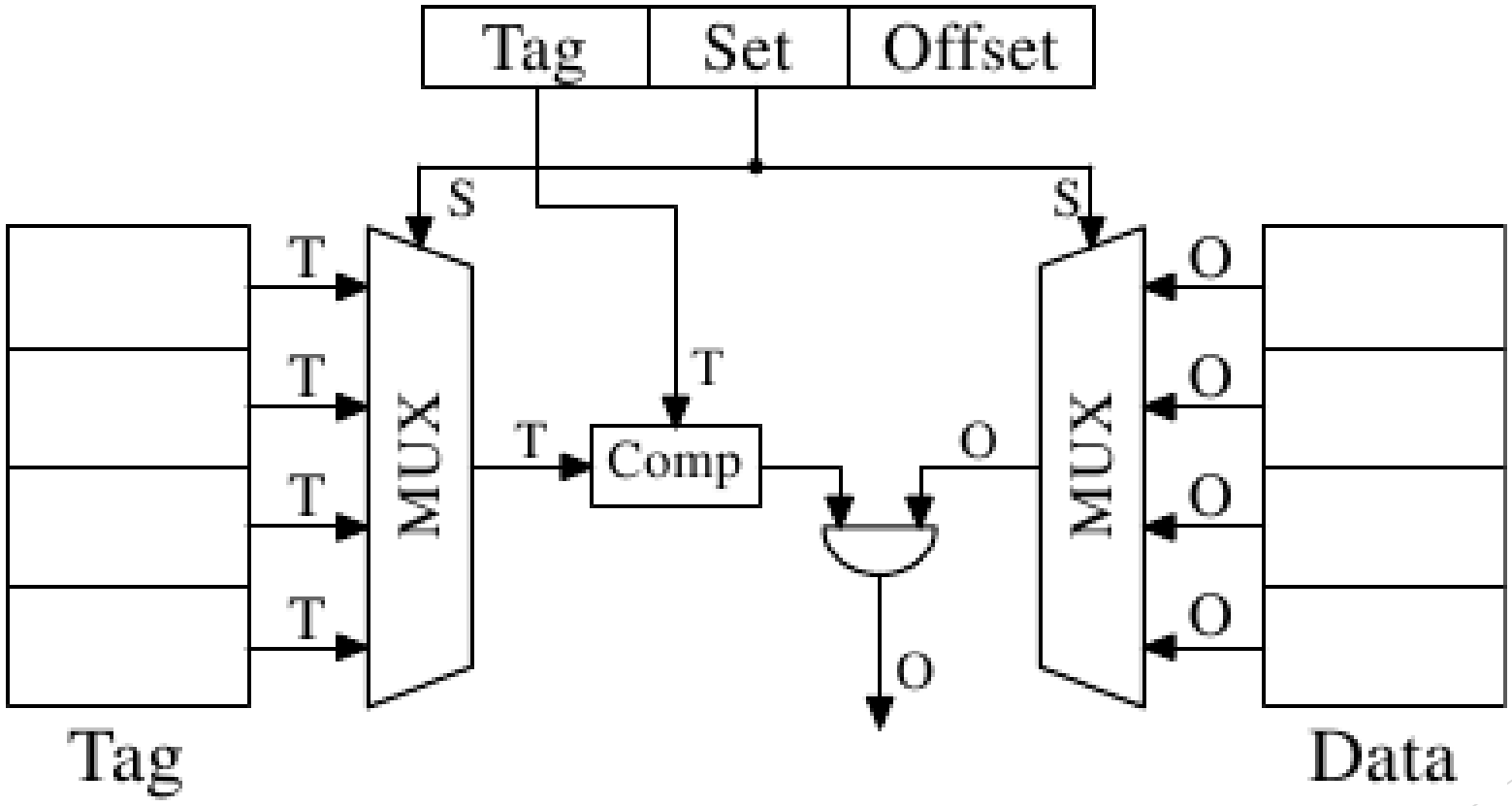


Fully associative cache

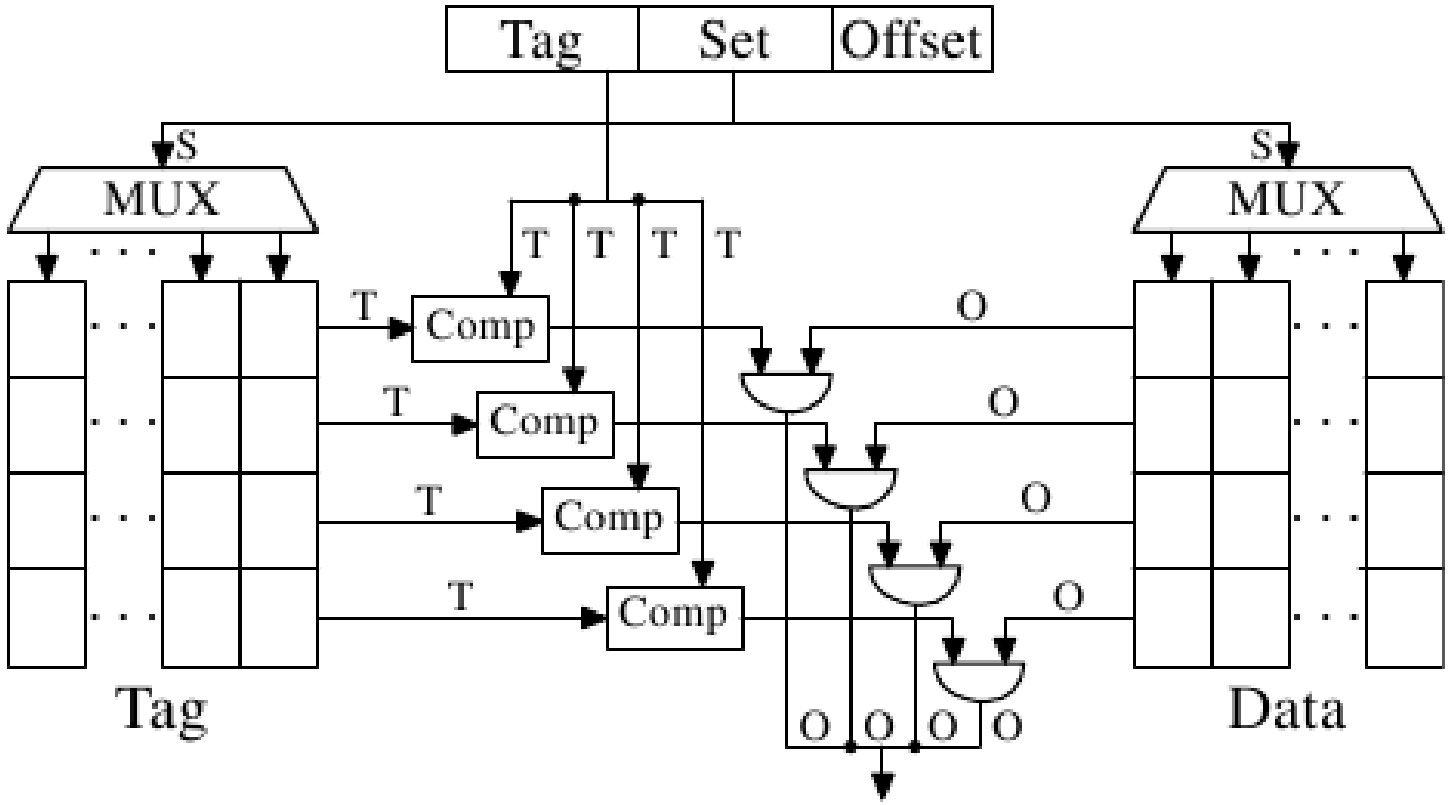


Comparator

Direct-mapped cache



Set associative cache

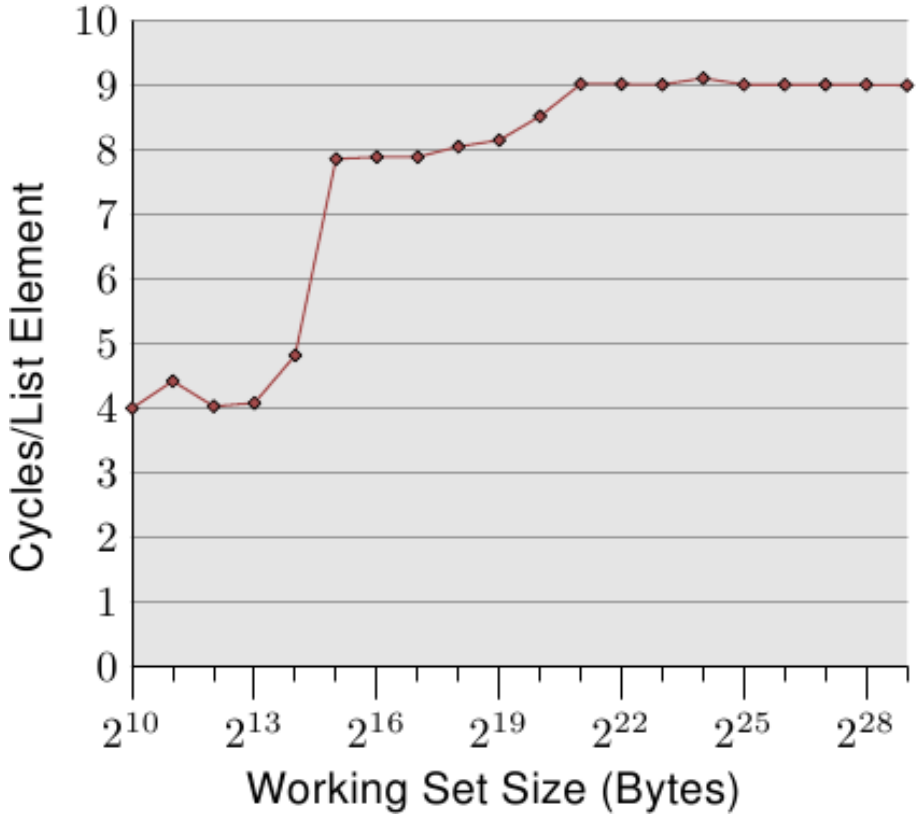


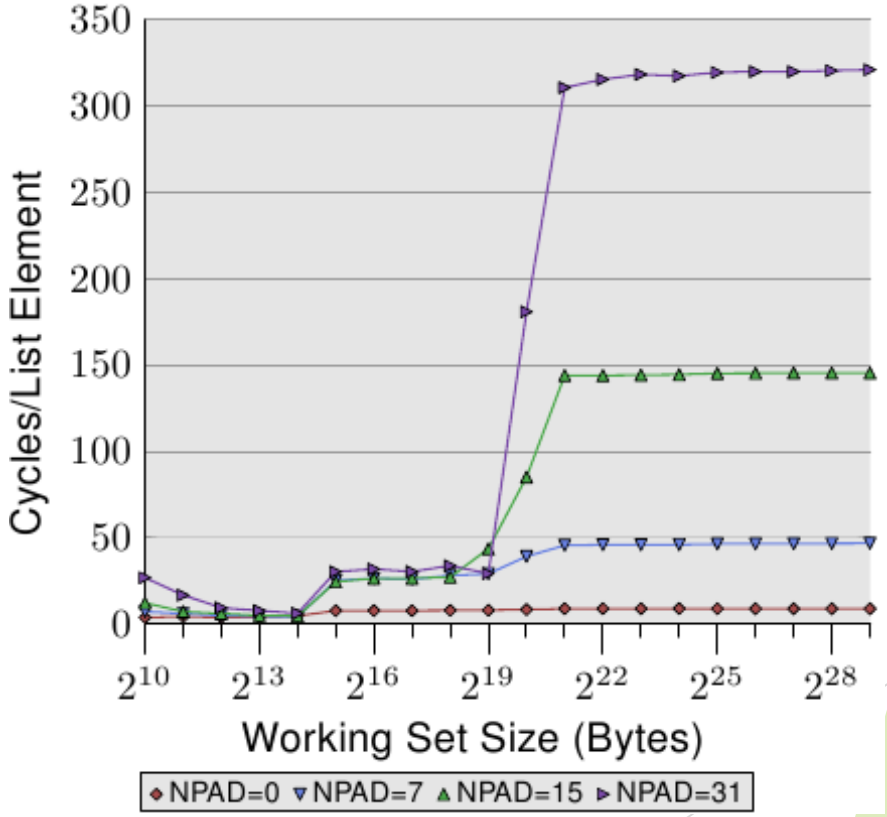
Cache effects

```
struct l {
        struct l *n;  // 64 bit
        long int pad[NPAD];
};
// 32 bytes cache line
```

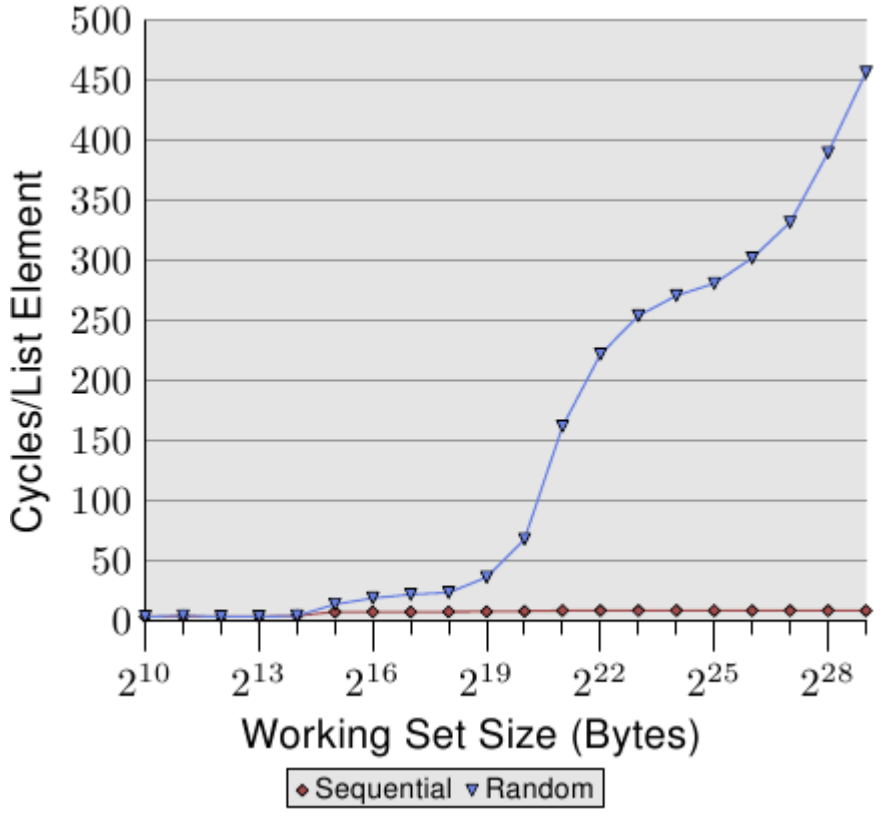
To where	Cycles
Register	<=1
L1d	~3
L2	~14
Main memory	~200

Cache effects





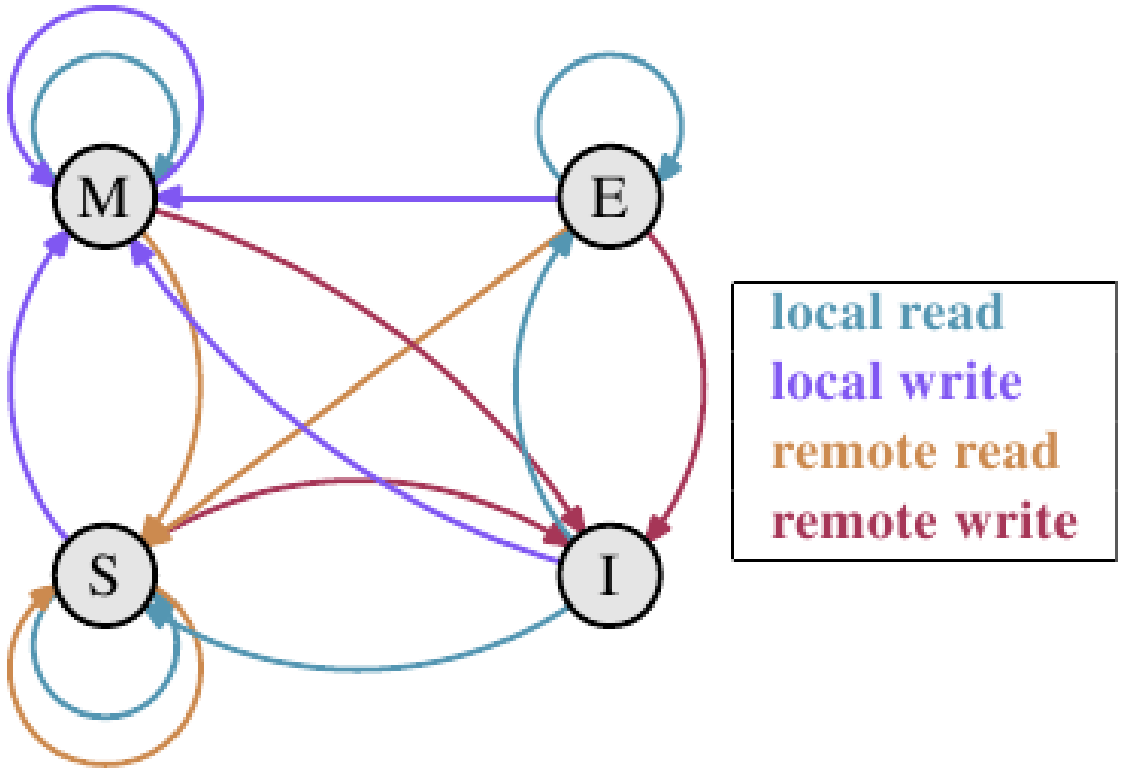
Cache effects



Write behavior

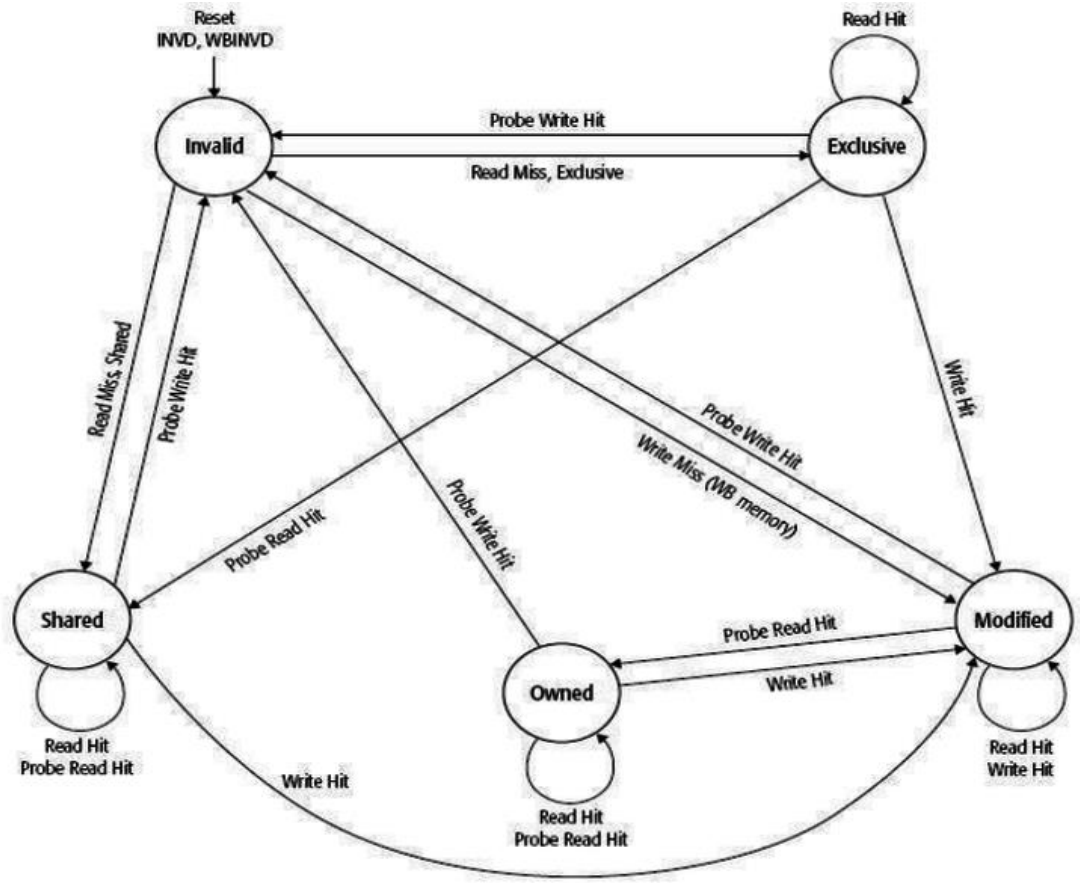
- Write-thorugh
- Write-back
- Write-combining
- Uncacheable

Multi-processor support

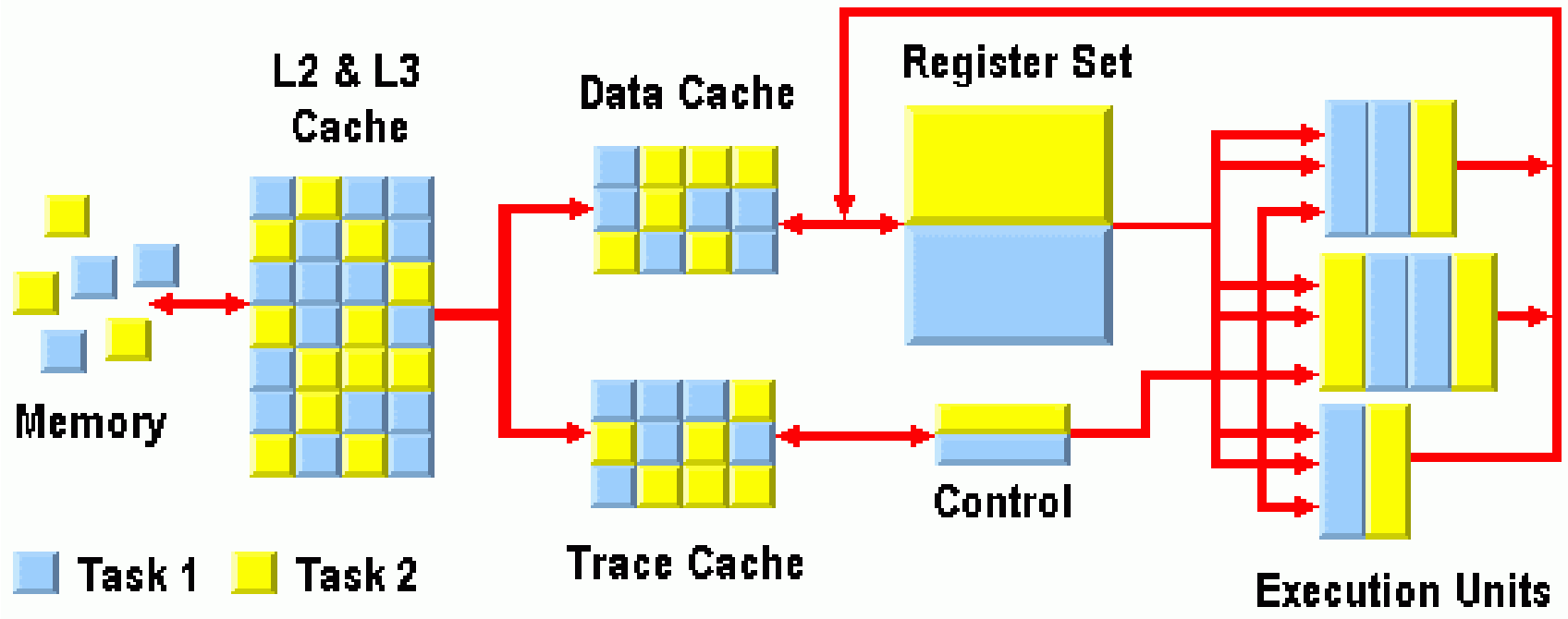


- Modified
- Exclusive
- Shared
- Invalid

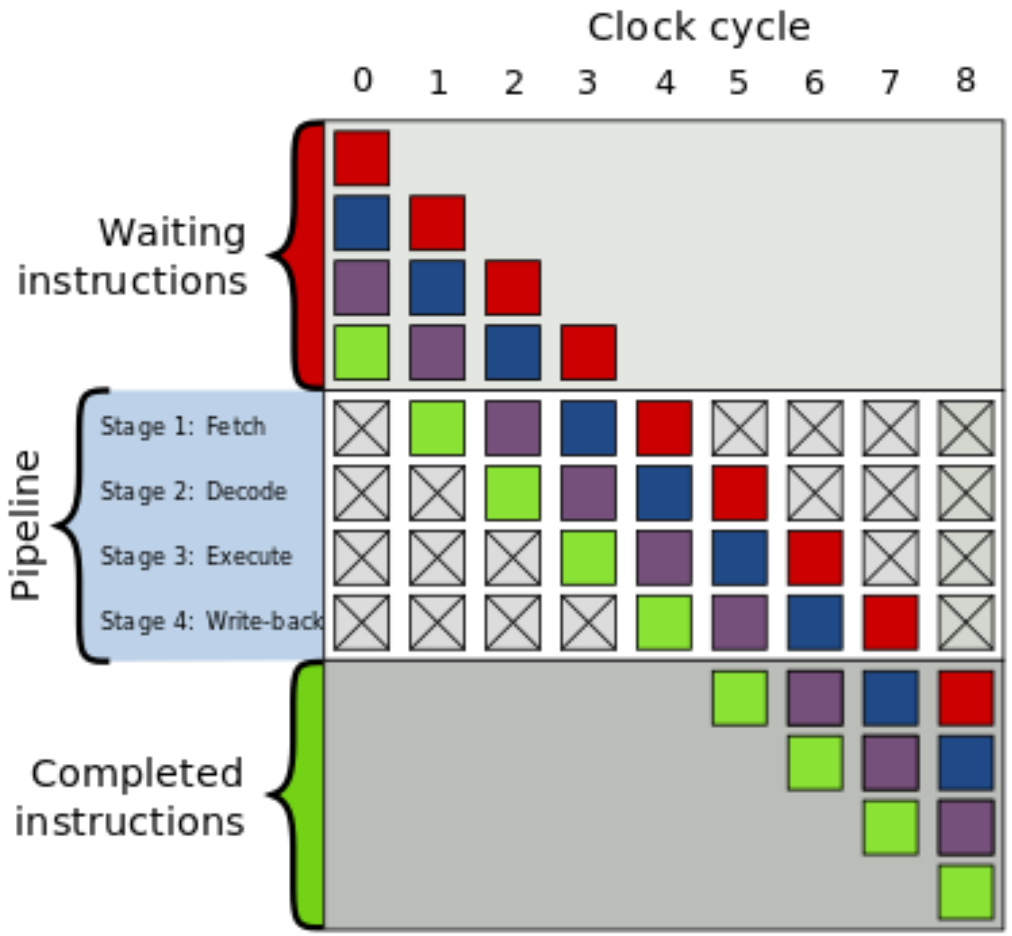
MOESI protocol



Hyper-threads



Instruction cache



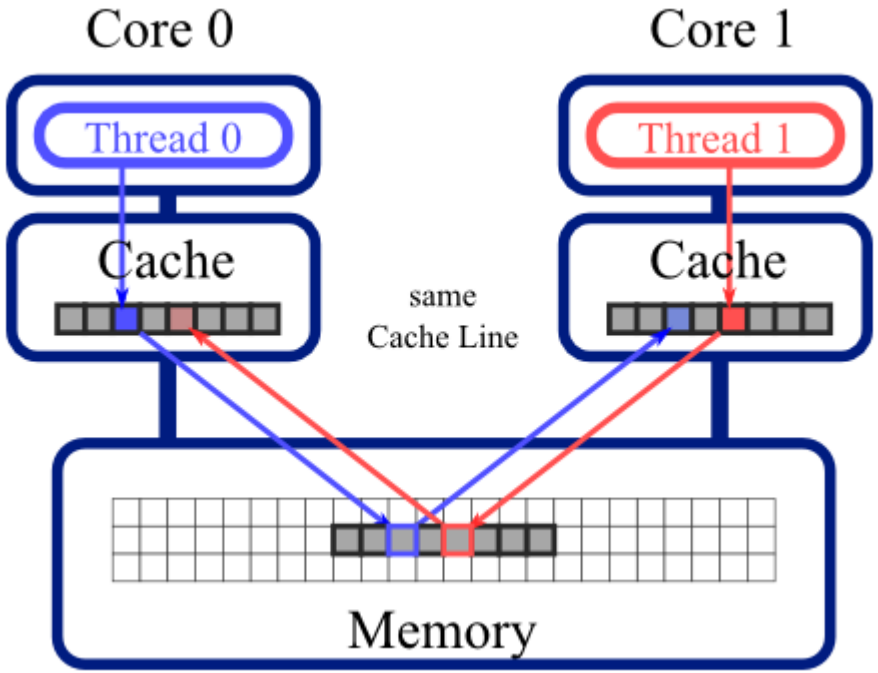
Meltdown

```
void f(char* p) {
    long t[256];
    // Evict cache
    if (complicated()) { //complicated() always false
        char c = *p;
        long c2 = t[c];
    }
    // Measure array access time
}
```

Spectre

```
// The attacker control x
// array1_size is not cached
// array1 is cached
if (x < array1_size) {
      y = array2[array1[x] * 256];
}
```

Concurrency optimizations



Optimizing level 1 data cache access

- Matrix multiplication:

```
for (i = 0; i < N; ++i)
      for (j = 0; j < N; ++j)
            for (k = 0; k < N; ++k)
                  res[i][j] += mul1[i][k] * mul2[k][j];
```

Relative: 100%

Optimizing level 1 data cache access

- Matrix multiplication:

```
double tmp[N][N];
for (i = 0; i < N; ++i)
      for (j = 0; j < N; ++j)
            tmp[i][j] = mul2[j][i];
for (i = 0; i < N; ++i)
      for (j = 0; j < N; ++j)
            for (k = 0; k < N; ++k)
                  res[i][j] += mul1[i][k] * mul2[k][j];
```

Relative: 23.4%

Optimizing level 1 data cache access

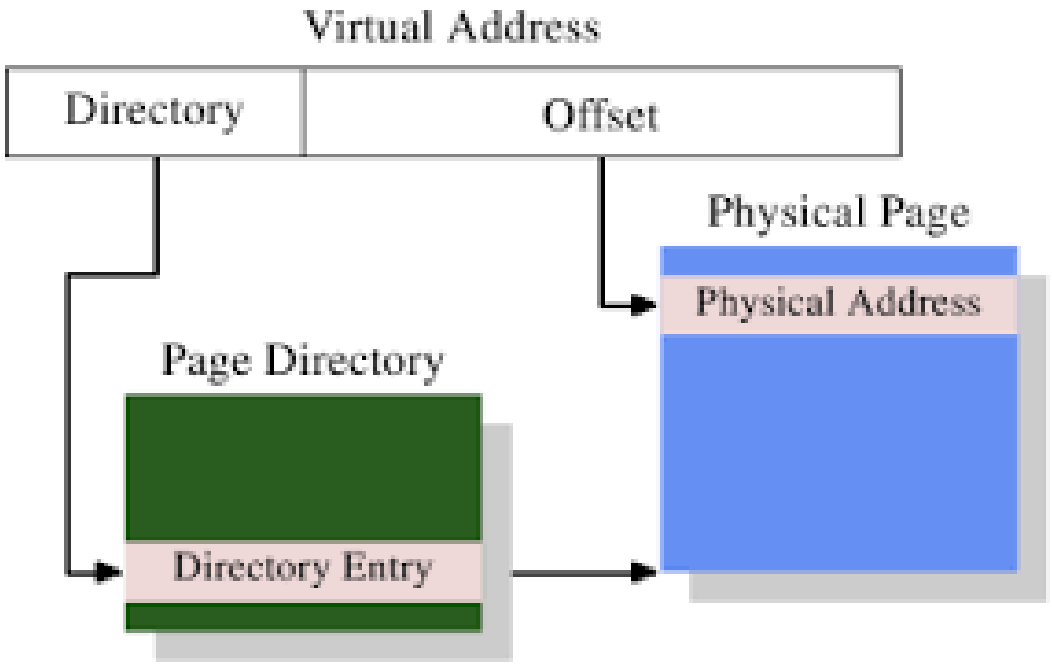
- Matrix multiplication:

```
#define SM (CLS / sizeof (double))
for (i = 0; i < N; i += SM)
      for (j = 0; j < N; j += SM)
            for (k = 0; k < N; k += SM)
                  for (i2 = 0, rres = &res[i][j],
                              rmul1 = &mul1[i][k]; i2 < SM;
                              ++i2, rres += N, rmul1 += N)
                        for (k2 = 0, rmul2 = &mul2[k][j];
                                    k2 < SM; ++k2, rmul2 += N)
                              for (j2 = 0; j2 < SM; ++j2)
                                    rres[j2] += rmul1[k2] * rmul2[j2];
```

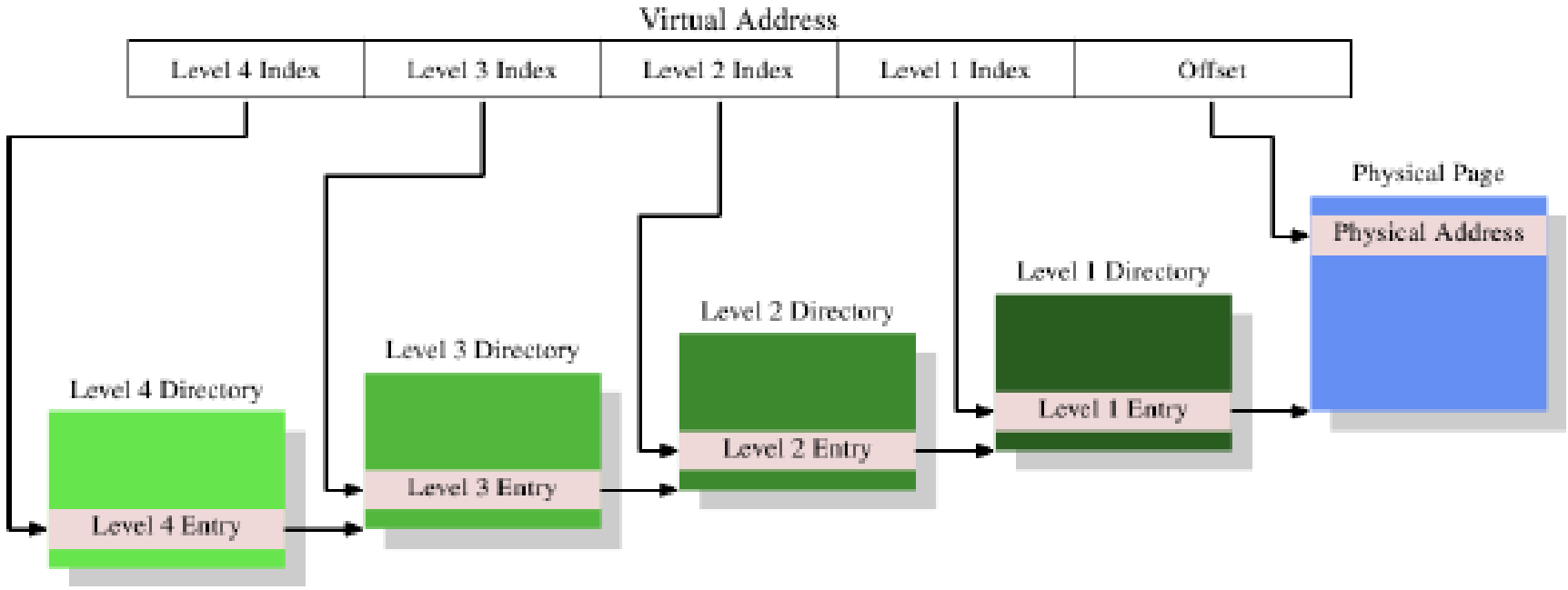
Relative: 17.3%

Virtual memory

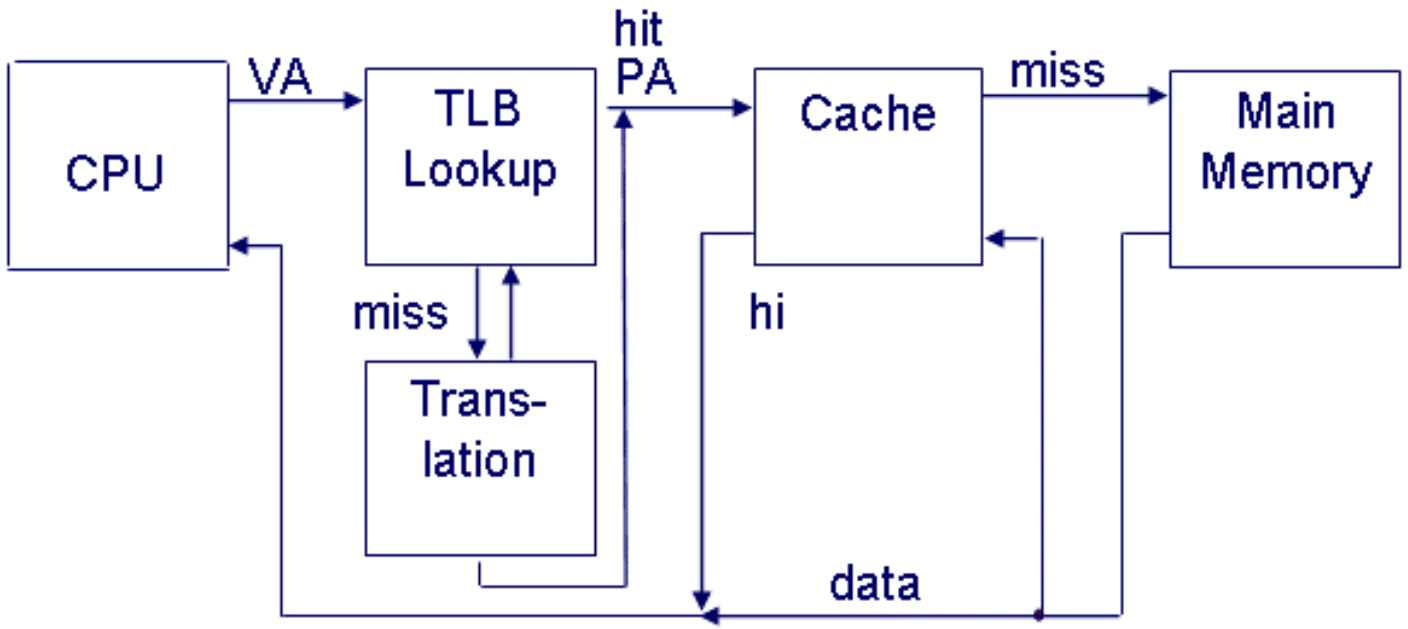
Page tables


Virtual Address
Directory
Offset
Physical Page
Physical Address
Page Directory
Directory Entry

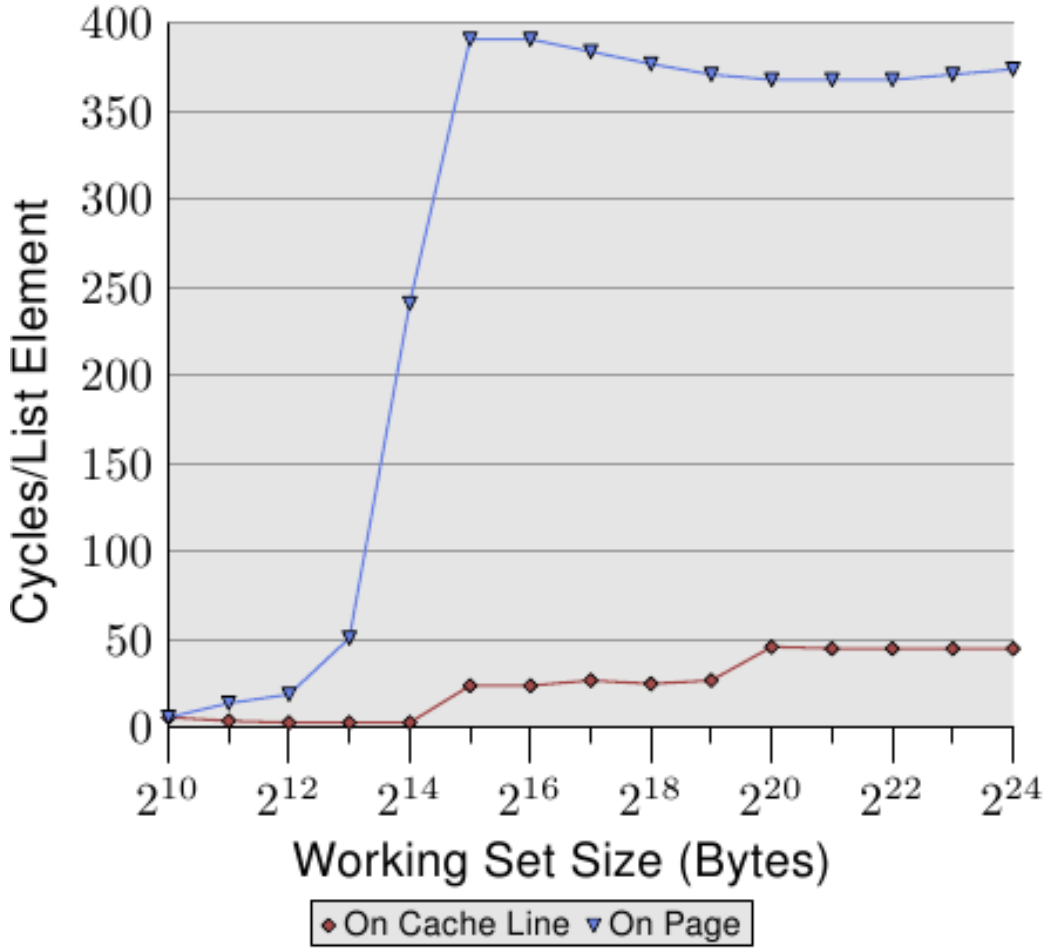
Multi-level page tables



Translation Look-aside Buffer(TLB)



TLB influence for sequential read



Comparator