Advanced Computer Architectures

Lecture – Memory subsystem – part one
- introduction, implementation, cache, virtual memory

Czech Technical University in Prague, Faculty of Electrical Engineering
Slides authors: Michal Štepanovský, update Pavel Píša

Lecture motivation from programmer POV?

Quick Quiz 1.: Is the result of both code fragments a same?

Quick Quiz 2.: Which of the code fragments is processed faster and why?

A:

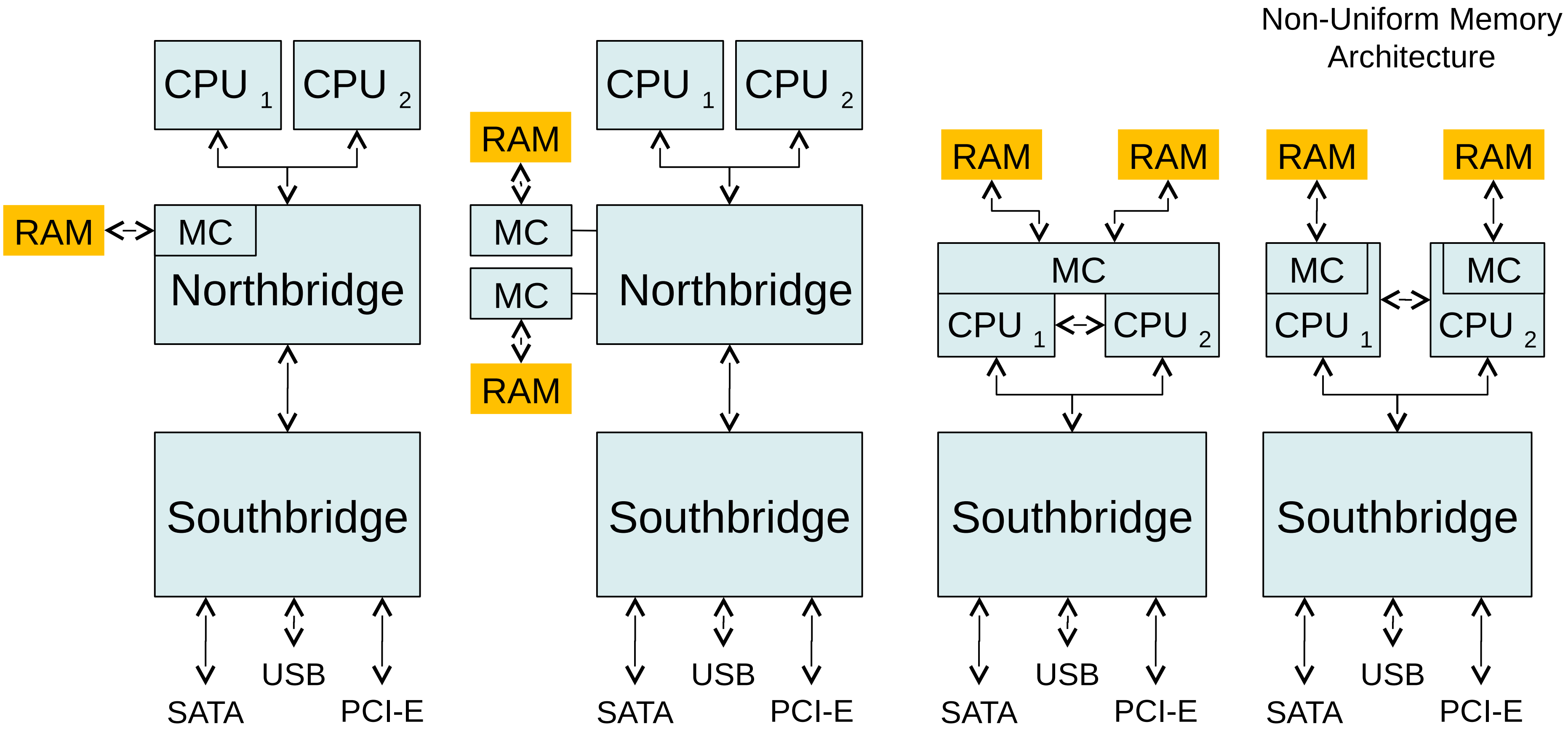
```
int matrix[M][N];
int i,j,sum=0;
…
for(i=0;i<M;i++)
  for(j=0;j<N;j++)
    sum+=matrix[i][j];
```

B:

```
int matrix[M][N];
int i,j,sum=0;
…
for(j=0;j<N;j++)
  for(i=0;i<M;i++)
    sum+=matrix[i][j];
```

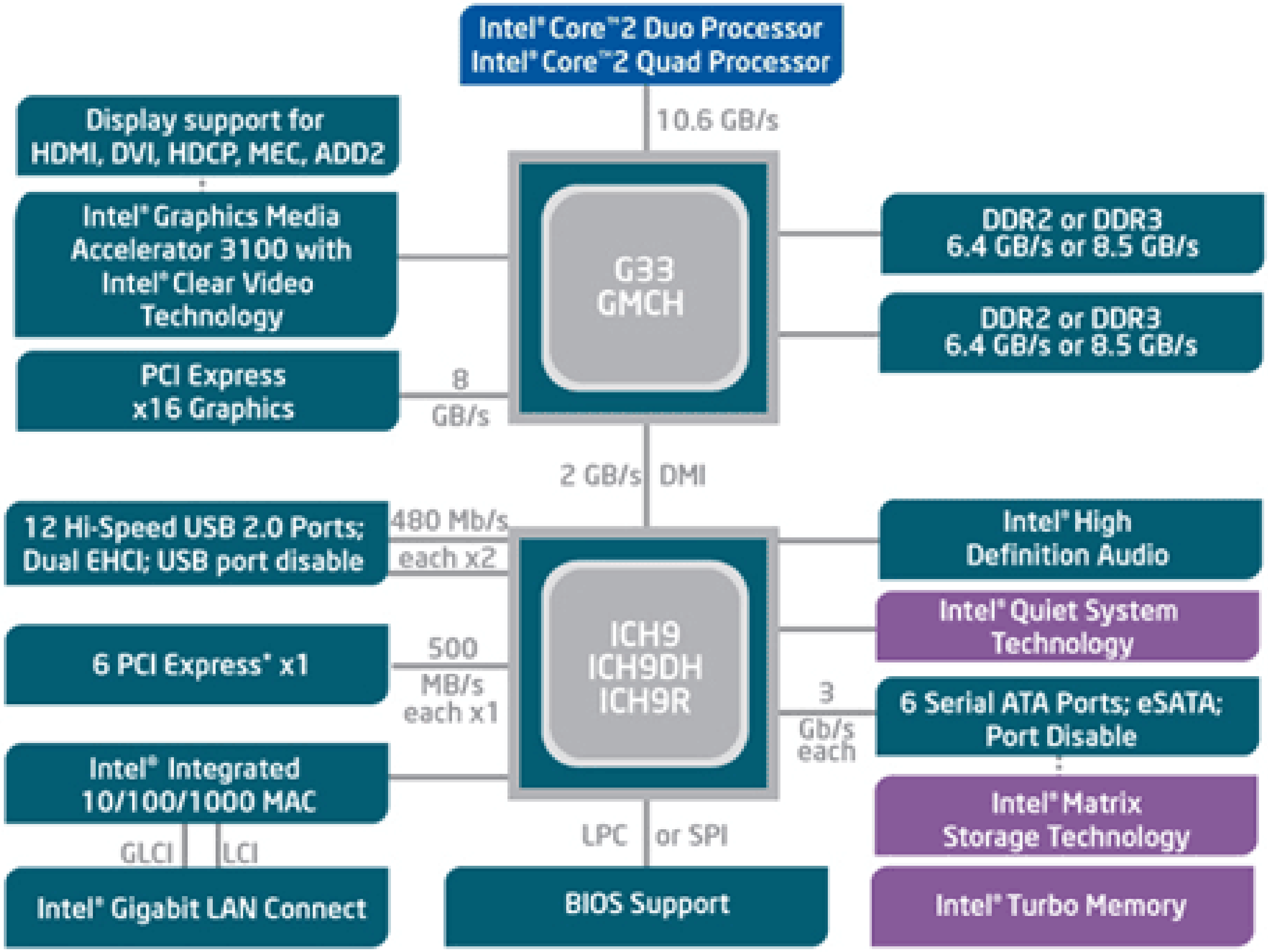
Is there a rule how to iterate over matrix element efficiently?

From UMA to NUMA development (even in PC segment)



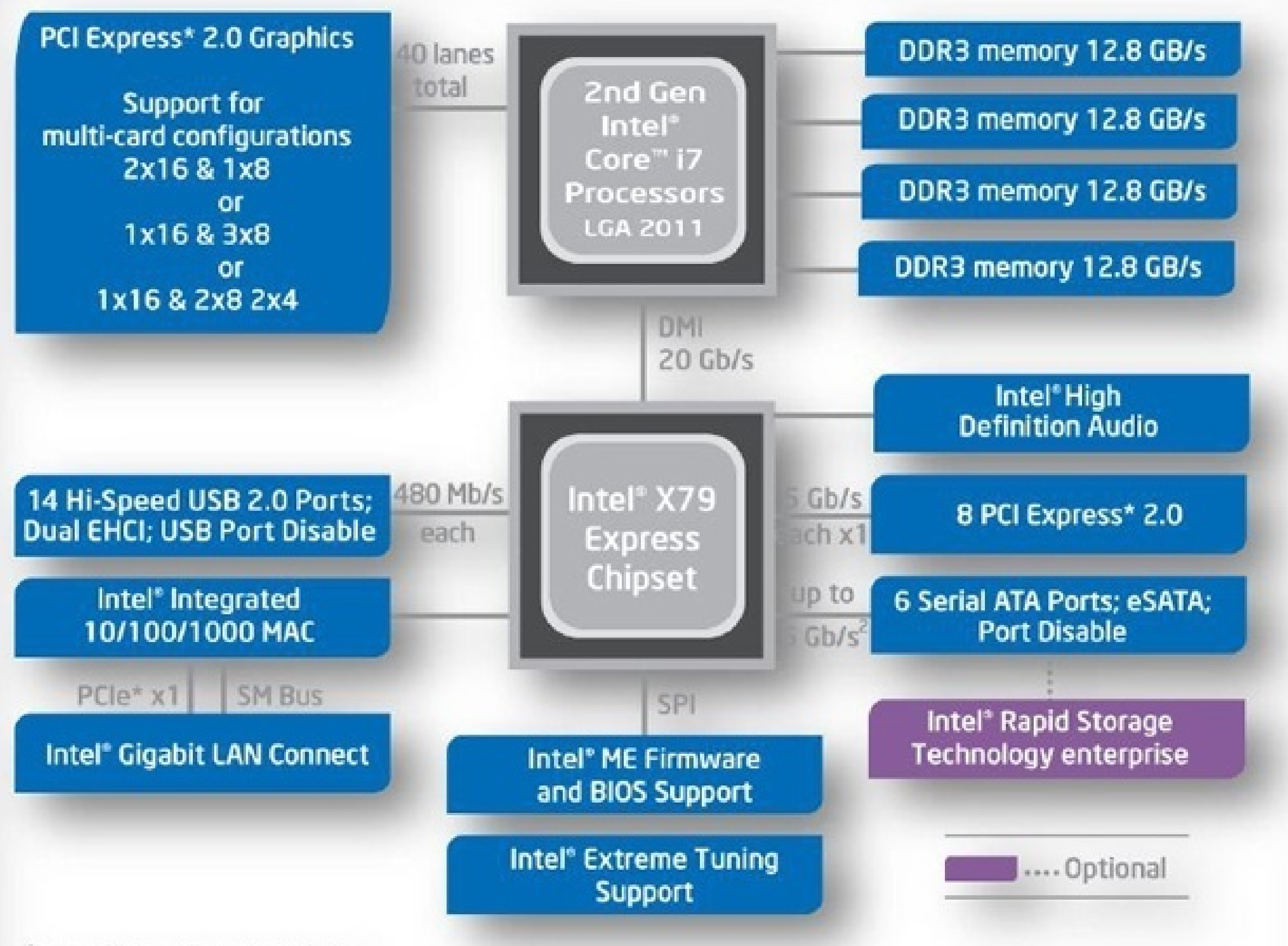
MC - Memory controller – contains circuitry responsible for SDRAM read and writes. It also takes care of refreshing each memory cell every 64 ms.

Intel Core 2 generation more detailed …


Intel® Core™2 Duo Processor
Intel® Core™2 Quad Processor
10.6 GB/s
Display support for
HDMI, DVI, HDCP, MEC, ADD2
Intel® Graphics Media Accelerator 3100 with Intel® Clear Video Technology
PCI Express x16 Graphics
8 GB/s
G33 GMCH
DDR2 or DDR3
6.4 GB/s or 8.5 GB/s
DDR2 or DDR3
6.4 GB/s or 8.5 GB/s
2 GB/s
DMI
12 Hi-Speed USB 2.0 Ports; Dual EHCI; USB port disable
480 Mb/s each x2
6 PCI Express* x1
500 MB/s each x1
Intel® Integrated 10/100/1000 MAC
GLCI
LCI
Intel® Gigabit LAN Connect
ICH9
ICH9DH
ICH9R
LPC
or SPI
BIOS Support
Intel® High Definition Audio
Intel® Quiet System Technology
3 Gb/s each
6 Serial ATA Ports; eSATA; Port Disable
Intel® Matrix Storage Technology
Intel® Turbo Memory

Northbridge became Graphics and Memory Controller Hub (GMCH)

Intel i3/5/7 generation more detailed …


PCI Express* 2.0 Graphics
Support for multi-card configurations 2x16 & 1x8 or 1x16 & 3x8 or 1x16 & 2x8 2x4
40 lanes total
2nd Gen Intel® Core™ i7 Processors LGA 2011
DDR3 memory 12.8 GB/s
DDR3 memory 12.8 GB/s
DDR3 memory 12.8 GB/s
DDR3 memory 12.8 GB/s
DMI 20 Gb/s
Intel® High Definition Audio
14 Hi-Speed USB 2.0 Ports; Dual EHCI; USB Port Disable
480 Mb/s each
Intel® X79 Express Chipset
5 Gb/s each x1
8 PCI Express* 2.0
Intel® Integrated 10/100/1000 MAC
up to 6 Gb/s²
6 Serial ATA Ports; eSATA; Port Disable
PCIe* x1
SM Bus
SPI
Intel® Gigabit LAN Connect
Intel® ME Firmware and BIOS Support
Intel® Rapid Storage Technology enterprise
Intel® Extreme Tuning Support
.... Optional

[1] Theoretical maximum bandwidth

[2] All SATA ports capable of 3 Gb/s. 2 ports capable of 6 Gb/s.

Intel® X79 Express Chipset Block Diagram

Even more details in following lectures

Memory subsystem – terms and definitions

- Memory address – fixed-length sequences of bits or index
- Data value – the visible content of a memory location

  Memory location can hold even more control/bookkeeping information

  - validity flag, parity and ECC bits etc.

- Basic memory parameters:
  - Access time – delay or latency between a request and the access being completed or the requested data returned
  - Memory latency – time between request and data being available (does not include time required for refresh and deactivation)
  - Throughput/bandwidth – main performance indicator. Rate of transferred data units per time.
  - Maximal, average and other latency parameters

Memory subsystem – terms and definitions

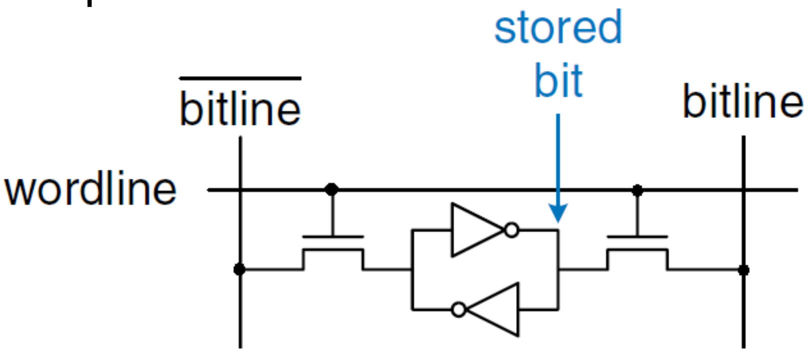
- Memory types RWM (RAM), ROM, FLASH
- Data retention time and conditions (volatile/nonvolatile)
- RAM memories implementations:
  SRAM (static), **DRAM** (dynamic).

- RAM = *Random Access Memory – memory with arbitrary address/random access*

memory type	**required transistors**	**area for 1 bit**	**data availability**	**latency**
SRAM	about 6	< 0,1 μm^2	all the time	< 1ns – 5ns
DRAM	1	< 0,001 μm^2	requires refresh	about 10 ns

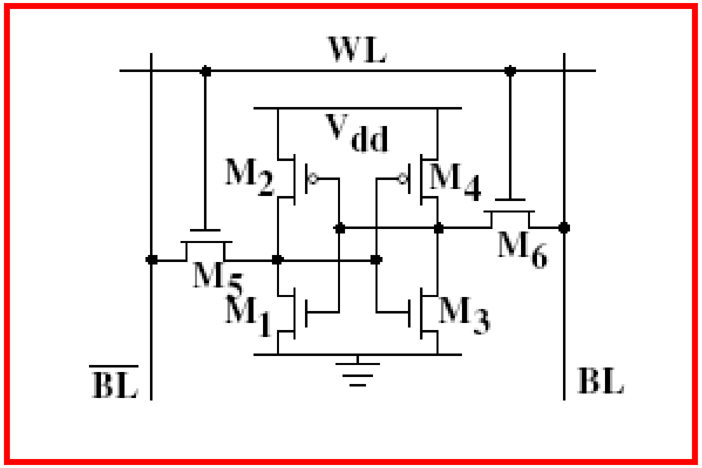
Usual SRAM chip and SRAM cell

Principle:

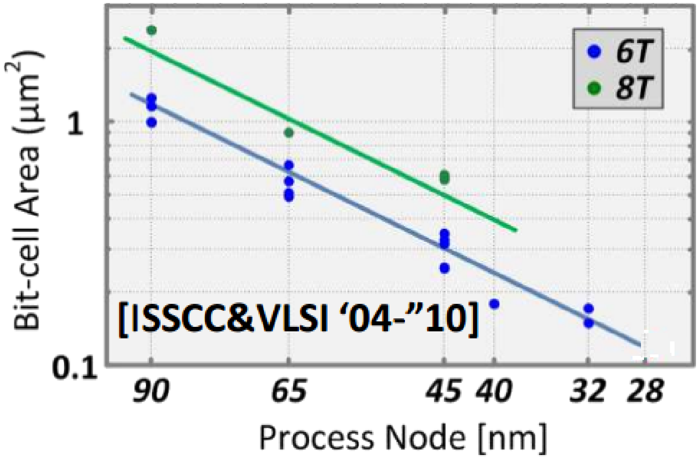


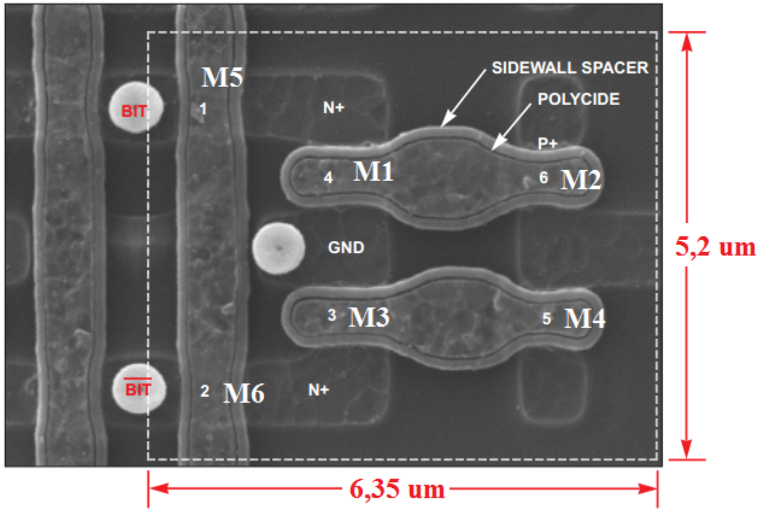
SRAM memory cell

CMOS technology



Area per memory cell:



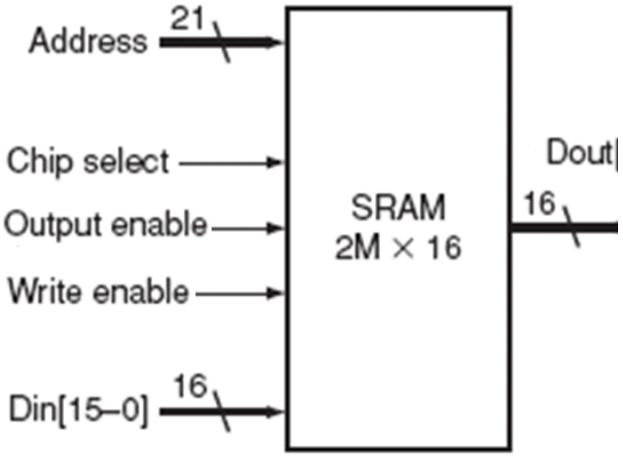


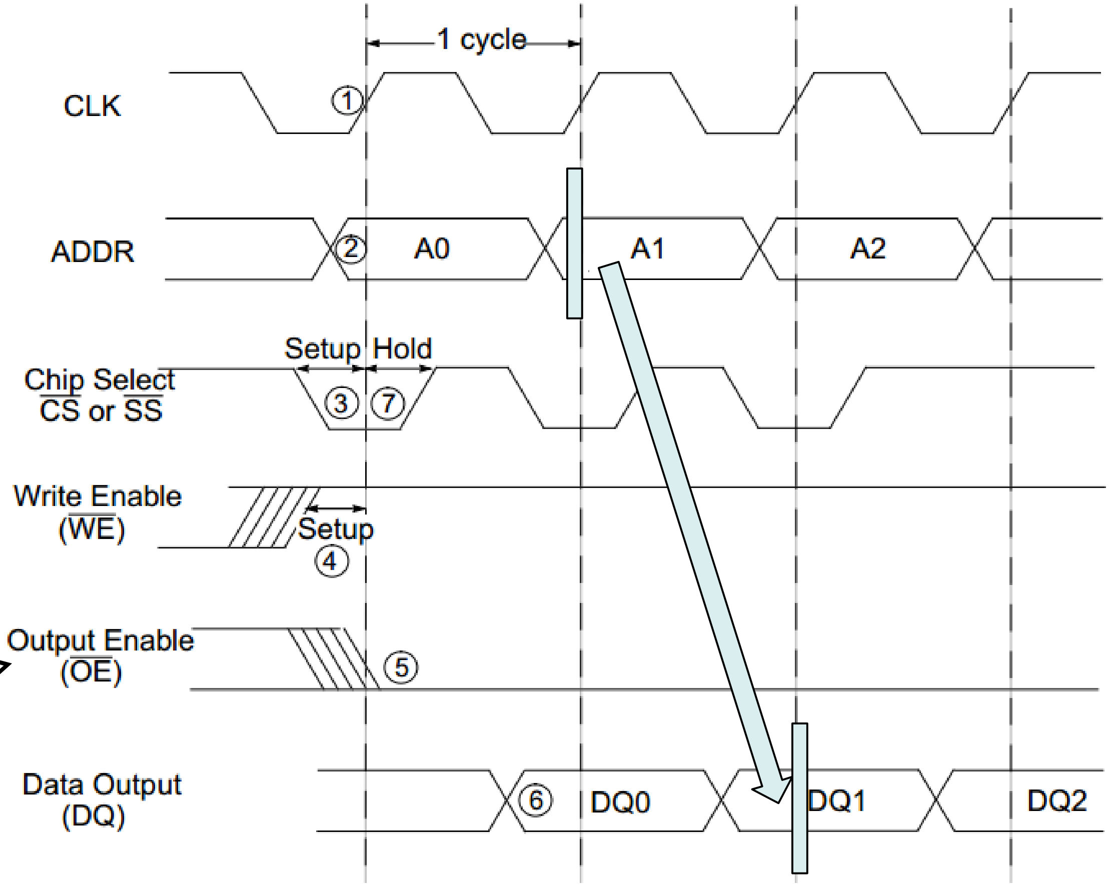
http://educypedia.karadimov.info/library/SEC08.PDF

Usual SRAM chip and SRAM cell

Common SRAM chip

Read example for synchronous case :



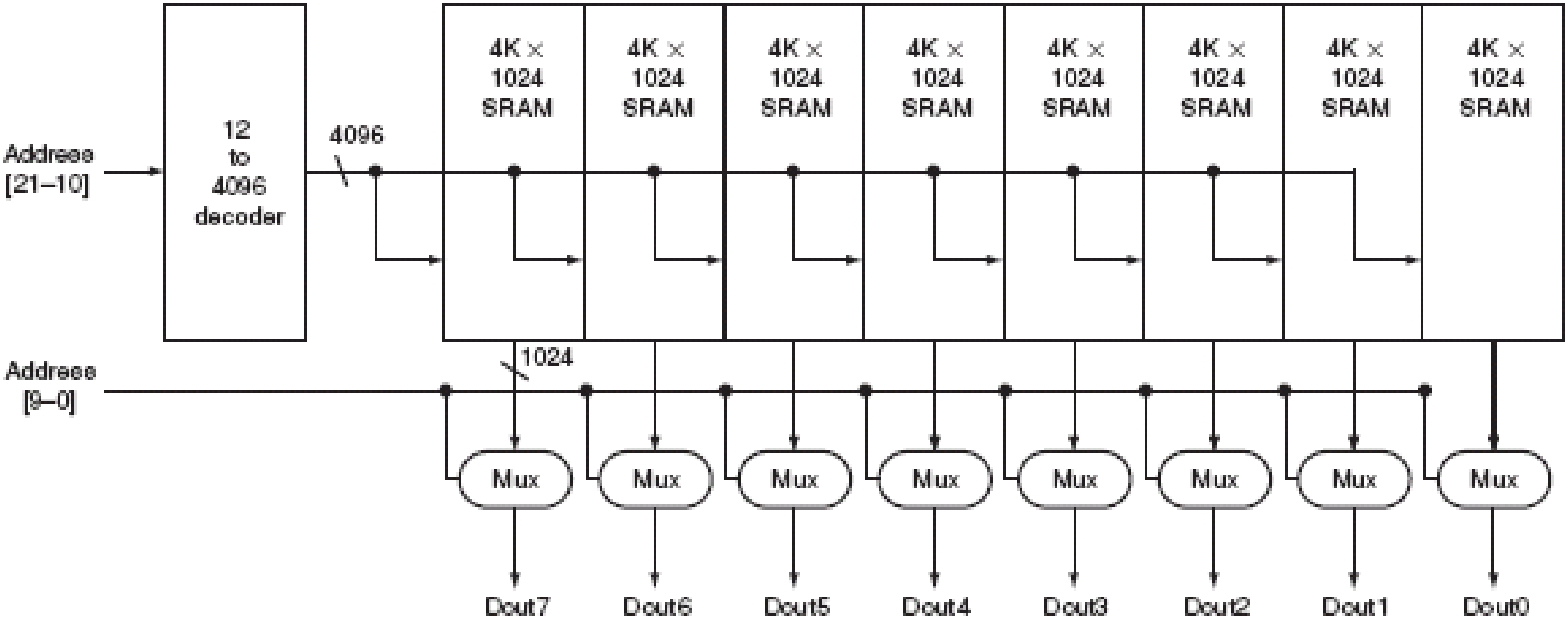


OE can be asynchronous

https://www.ece.cmu.edu/~ece548/localcpy/sramop.pdf

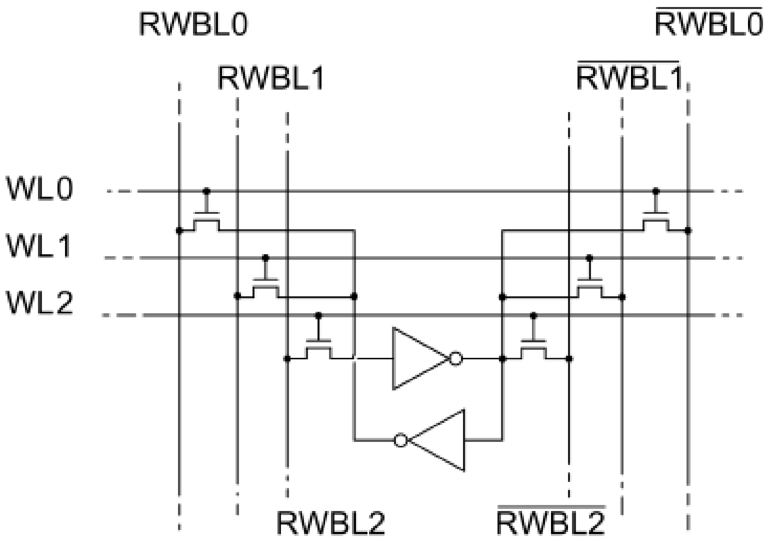
Usual SRAM chip and SRAM cell

Larger memory?

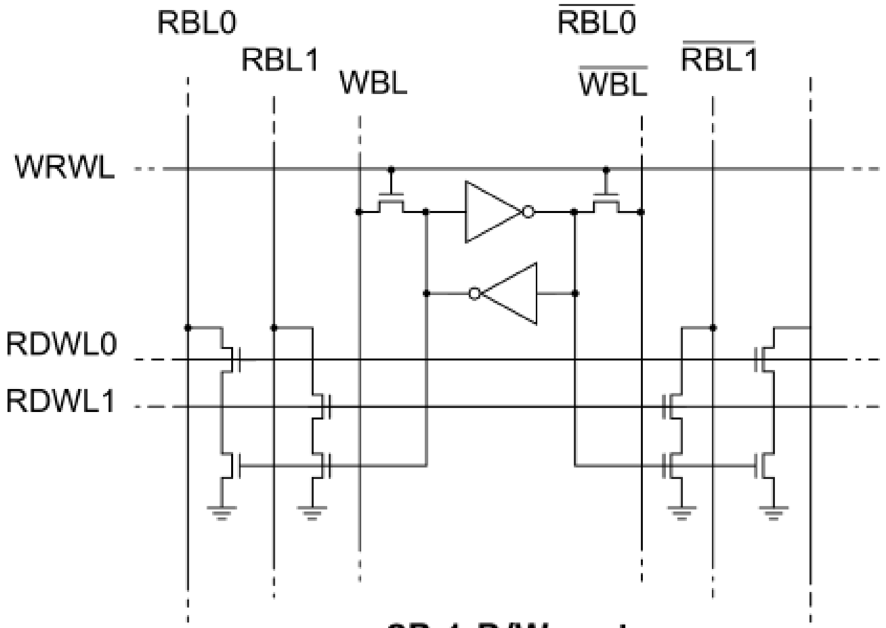


Multiport cache?

- A must for superscalar CPU
- Required when cache is shared between CPUs as well


RWBL0
$\overline{\text{RWBL0}}$
RWBL1
$\overline{\text{RWBL1}}$
WL0
WL1
WL2
RWBL2
$\overline{\text{RWBL2}}$

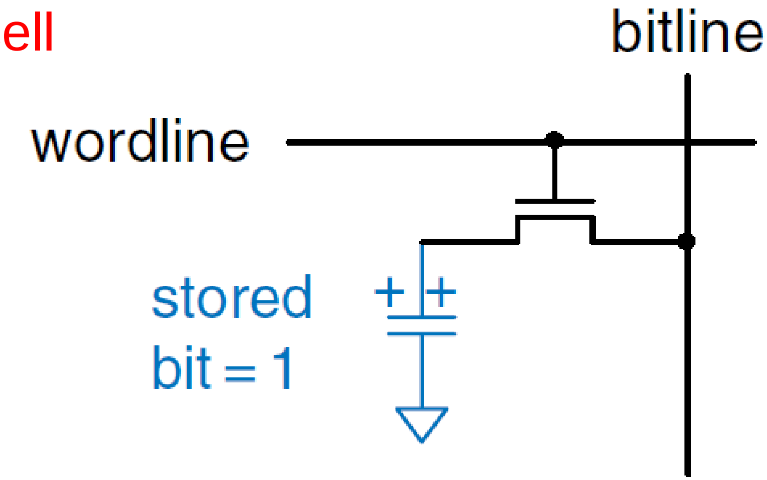
3 R/W ports

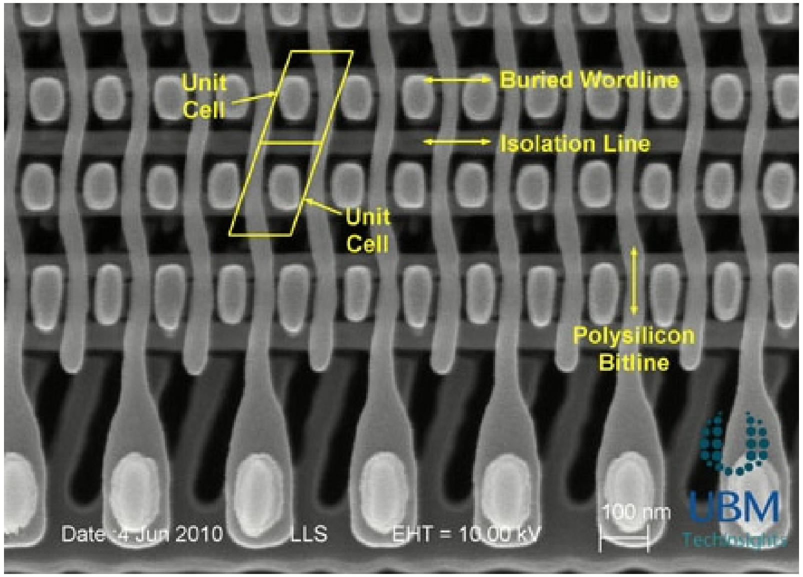


2R 1 R/W port

Detail of dynamic memory cell

Single transistor dynamic memory bit cell



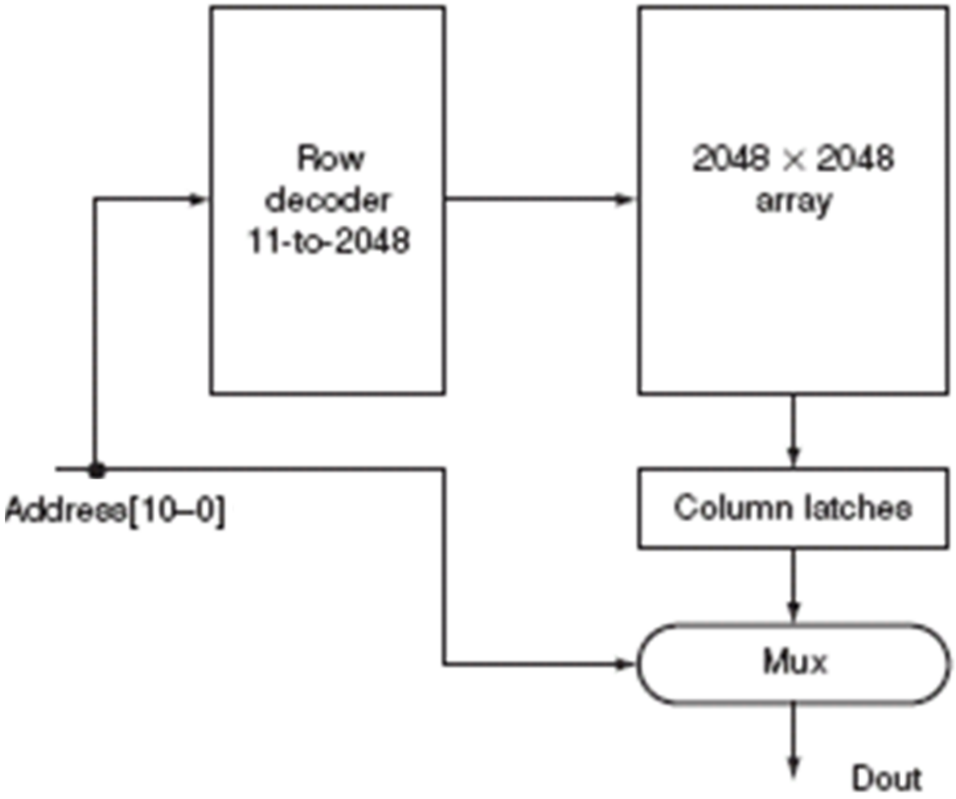


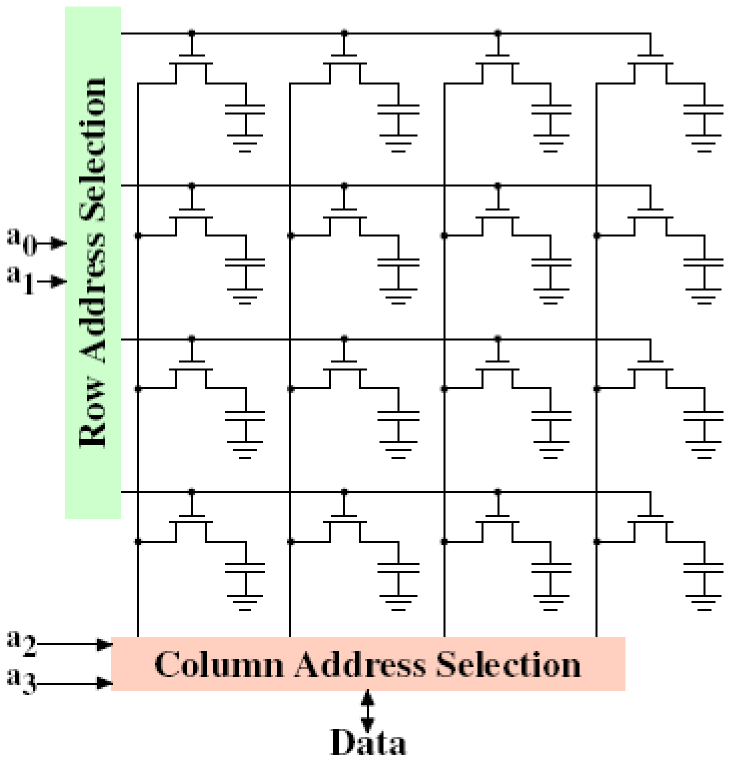
The nMOS transistor forms a switch that connects (or not) the capacitor to the "bitline" wire. The connection is controlled by a "wordline" wire.

The process of reading discharges the capacitor same as the current leakage in time. Therefore, its state must be renewed. Refresh - required working phase of dynamic memory. Negatively affects (prolongs) the average access time.

Photo source: http://www.eetimes.com/document.asp?doc_id=1281315

Internal architecture of the DRAM memory chip

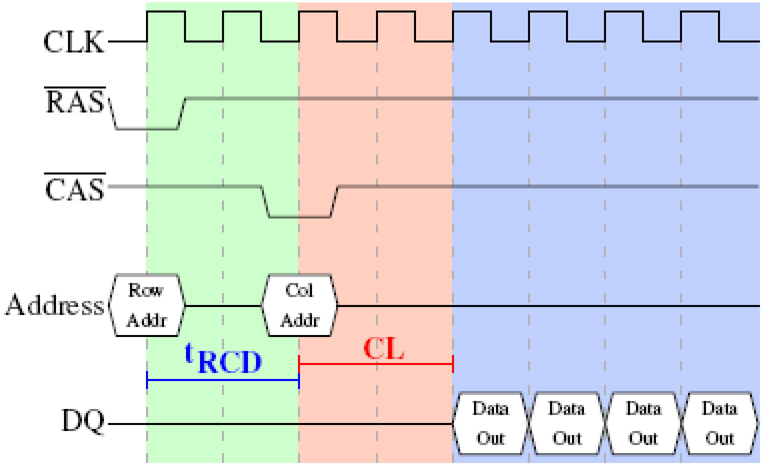




This 4M × 1 DRAM is internally realized as an 2048x2048 array of 1b memory cells

History of DRAM chips development

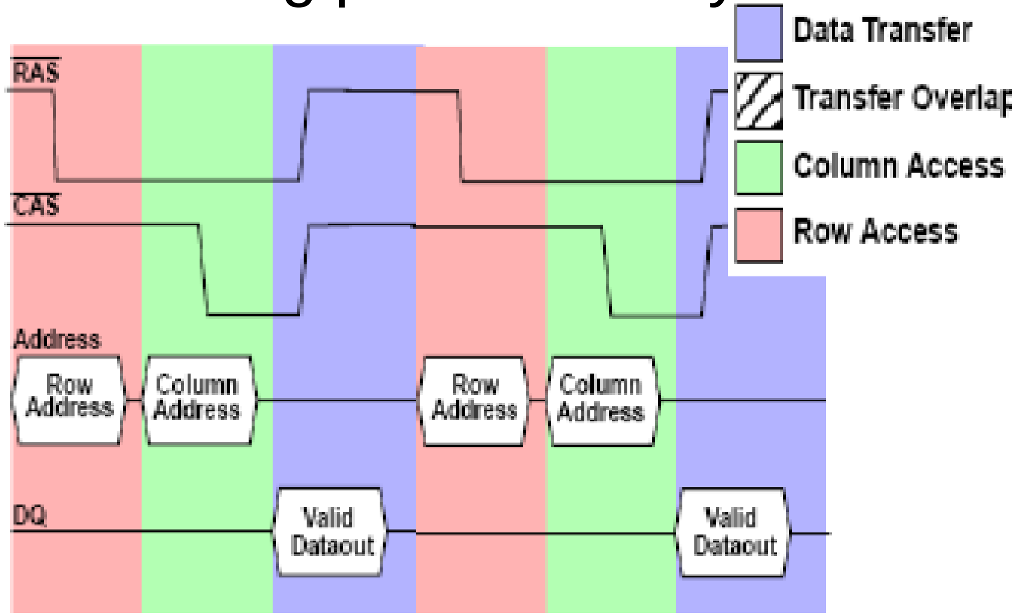
Year	Capacity	Price[$]/GB	Access time [ns]
1980	64 Kb	1 500 000	250
1983	256 Kb	500 000	185
1985	1 Mb	200 000	135
1989	4 Mb	50 000	110
1992	16 Mb	15 000	90
1996	64 Mb	10 000	60
1998	128 Mb	4 000	60
2000	256 Mb	1 000	55
2004	512 Mb	250	50
2007	1 Gb	50	40



RAS – Row Address Strobe,
CAS – Column Address Strobe

Old school DRAM – asynchronous access

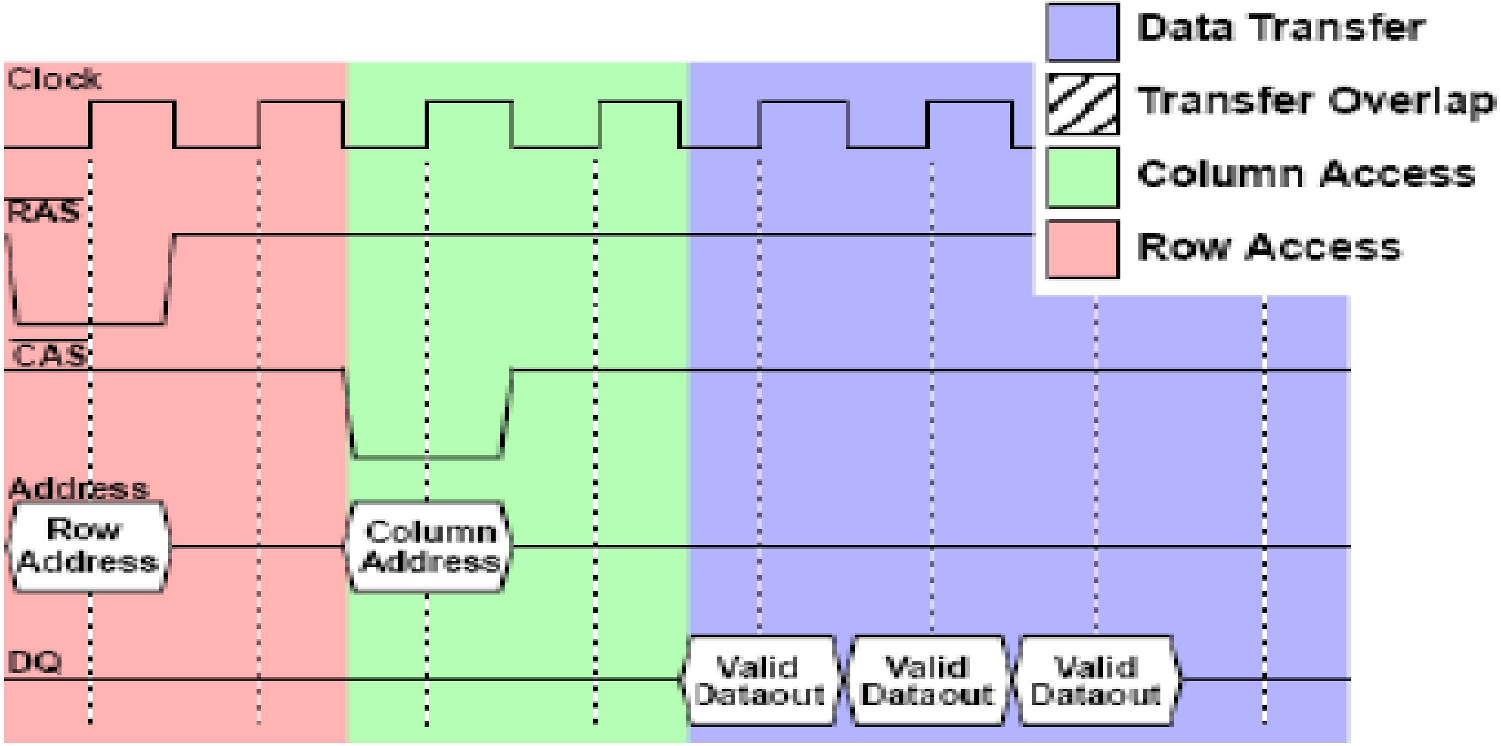
- The address is transferred in two phases – reduces number of chip module pins and is natural for internal DRAM organization
- This method is preserved even for today chips even that more pins/balls are no so big problem today



RAS – Row Address Strobe,
CAS – Column Address Strobe

SDRAM – end of 90-ties – synchronous DRAM

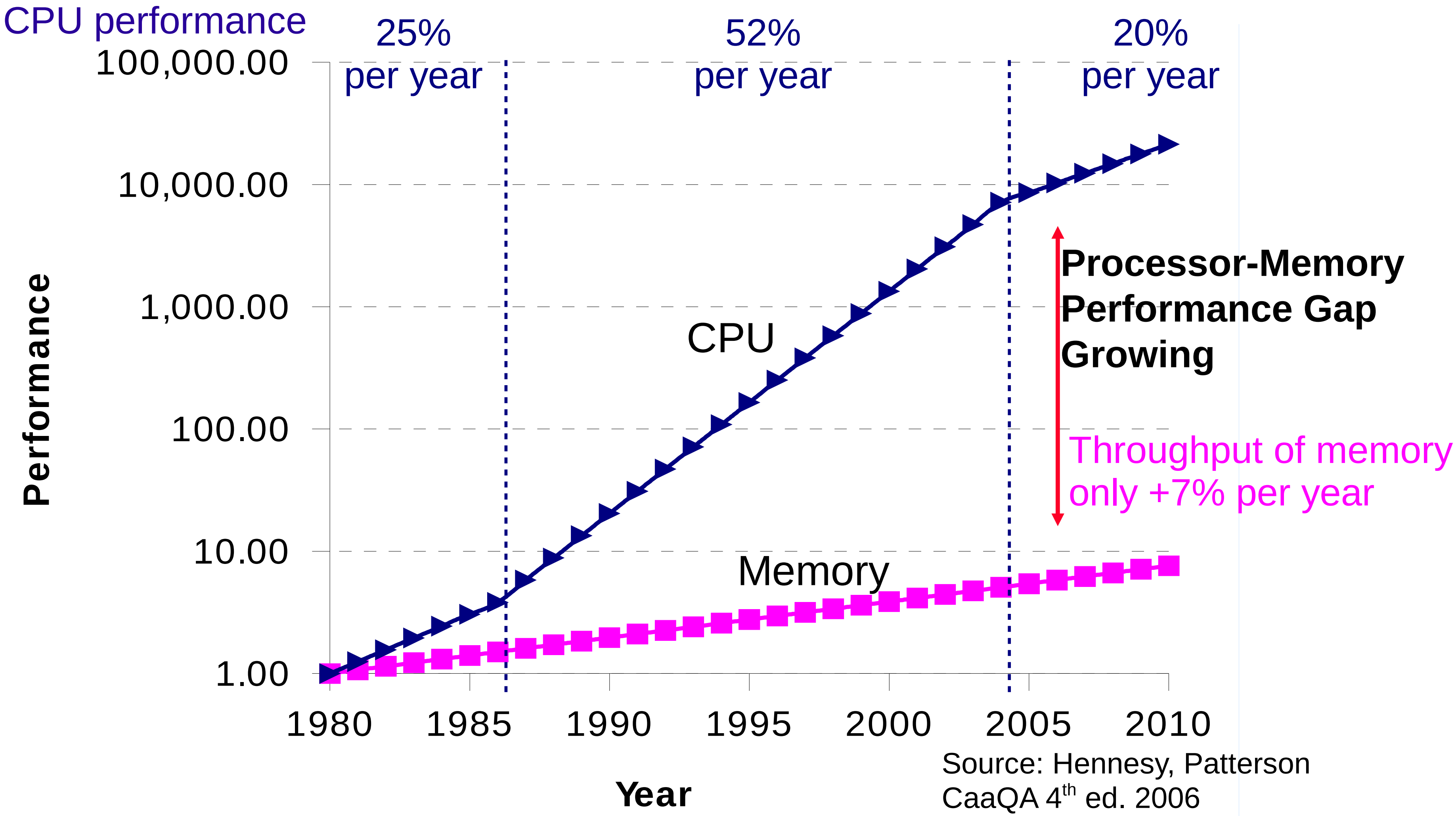
- SDRAM chip is equipped by counter that can be used to define continuous block length (burst) which is read together



SDRAM – the most widely used main memory technology

- SDRAM – clock frequency up to 100 MHz, 2.5V.
- DDR SDRAM – data transfer at both CLK edges, 2.5V.
- DDR2 SDRAM – lower power consumption 1.8V, frequency up to 400 MHz.
- DDR3 SDRAM – even lower power consumption at 1.5V, frequency up to 800 MHz.
- DDR4 SDRAM …
- There are also other dynamic memory types, I.e. RAMBUS, that use entirely different concept
- All these innovations are focused mainly on throughput, not on the random access latency.

Memory and CPU speed disproportion – Moore's law



Source: Hennesy, Patterson
CaaQA 4th ed. 2006

Bubble sort – algorithm example from seminaries

```
int array[5]={5,3,4,1,2};
int main()
{
    int N = 5,i,j,tmp;
    for(i=0; i<N; i++)
        for(j=0; j<N-1-i; j++)
            if(array[j+1]<array[j])
            {
                tmp = array[j+1];
                array[j+1] = array[j];
                array[j] = tmp;
            }
    return 0;
}
```

What we can consider and expect from our programs?

Think about some typical data access patterns and execution flow.

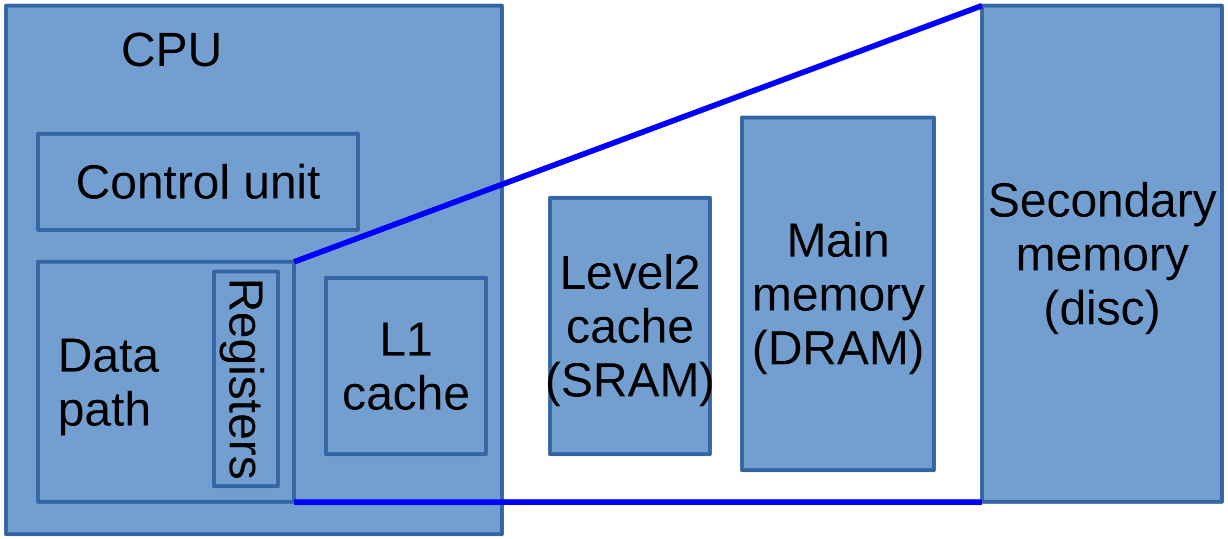
Memory hierarchy – principle of locality

- Programs/processes access a **small proportion** of their address space at any given instant of time
- **Temporal locality**
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, instruction variables
- **Spatial locality**
  - Items near those accessed recently are likely to be accessed soon.
  - E.g., sequential instruction access (program memory), array data (data memory).

Memory hierarchy introduced based on locality

- The solution to resolve capacity and speed requirements is to build address space (data storage in general) as hierarchy of different technologies.
- Store input/output data, program code and its runtime data on large and cheaper secondary storage (hard disk)
- Copy recently accessed (and nearby) items from disk to smaller DRAM based main memory (usually under operating system control)
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory (cache) attached to CPU (hidden memory, transactions under HW control), optionally, tightly coupled memory under program's control
- Move currently processed variables to CPU registers (under machine program/compiler control)

Contemporary price/size examples



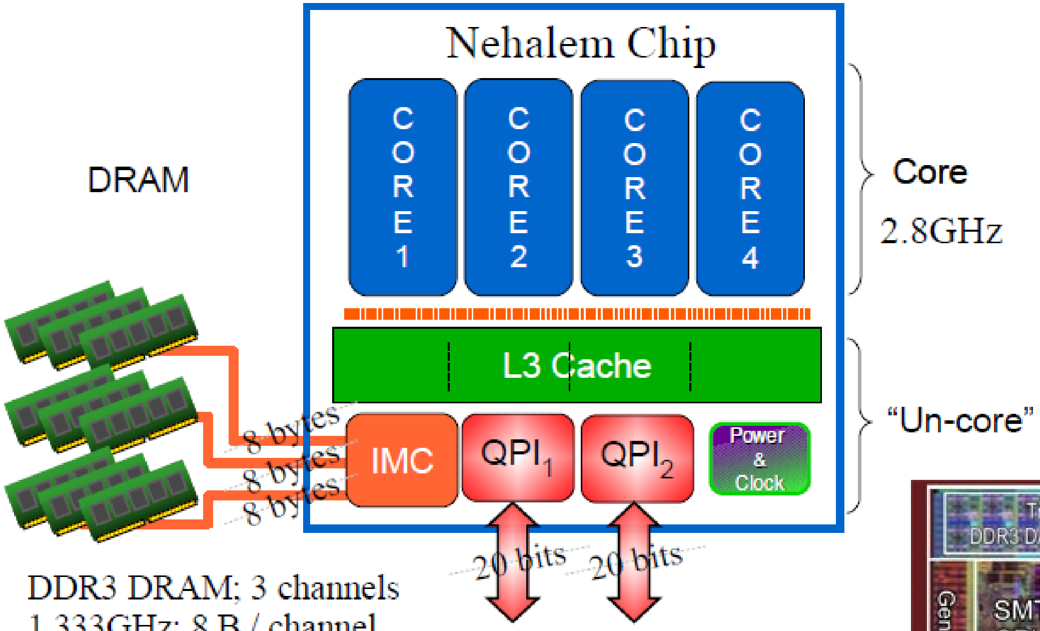
Type/ Size	L1 32kB	Sync SRAM	DDR3 16 GB	HDD 3TB
Price	10 kč/kB	300 kč/MB	123 kč/GB	1 kč/GB
Speed/ throughput	0.2...2ns	0.5...8 ns/word	15 GB/sec	100 MB/sec

Some data can be available in more copies (consider levels and/or SMP).
Mechanisms to keep consistency required if data are modified.

Mechanism to lookup demanded information?

- According to the address and other management information (data validity flags etc).
- The lookup starts at the most closely located memory level (local CPU L1 cache).
- Requirements:
  - Memory consistency/coherency.
- Used means:
  - Memory management unit to translate virtual address to physical and signal missing data on given level.
  - Mechanisms to free (swap) memory locations and migrate data between hierarchy levels
- Hit (data located in upper level – fast), miss (copy from lower level required)

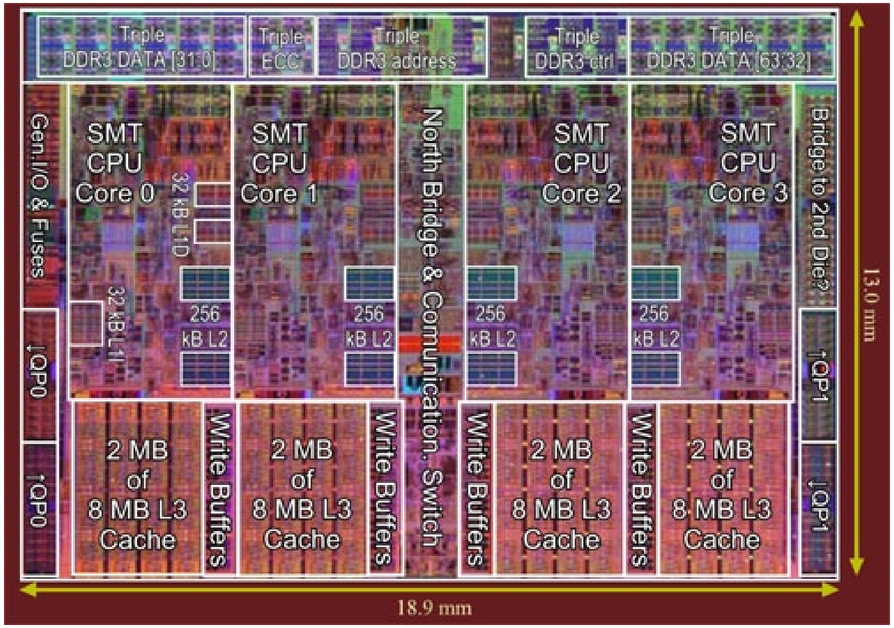
Example of the CPU with three-level cache
Harvard architecture L1 cache - Intel Nehalem



DDR3 DRAM; 3 channels
1.333GHz; 8 B / channel
31.992 GB/s aggregate
7.998 GB/s / core

Intel® QPI point-to-point link
6.4 GT/s; full-duplex;
12.8 GiB/s + **12.8**GiB/s

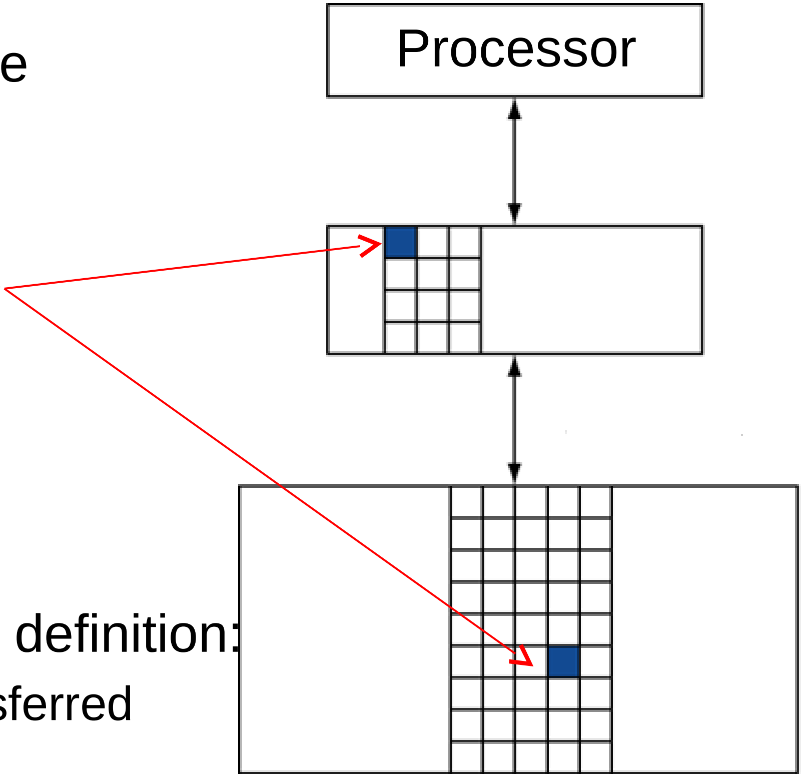
- IMC: integrated memory controller with 3 DDR3 memory channels,
- QPI: Quick-Path Interconnect ports
- Compare sizes of caches at each level!!!



Solution of memory and CPU speed disproportion? Cache.

Terminology definitions for cache memory

- **Cache hit** corresponds to situation when search location **is found** in the cache.
- **Cache miss**, opposite. **Not dound**.
- **Cache line** or **Cache block** – basic unit/size which is copied to/from memory.
- Cache block size from 8B to 1KB, typically 32/64/128B in practice.

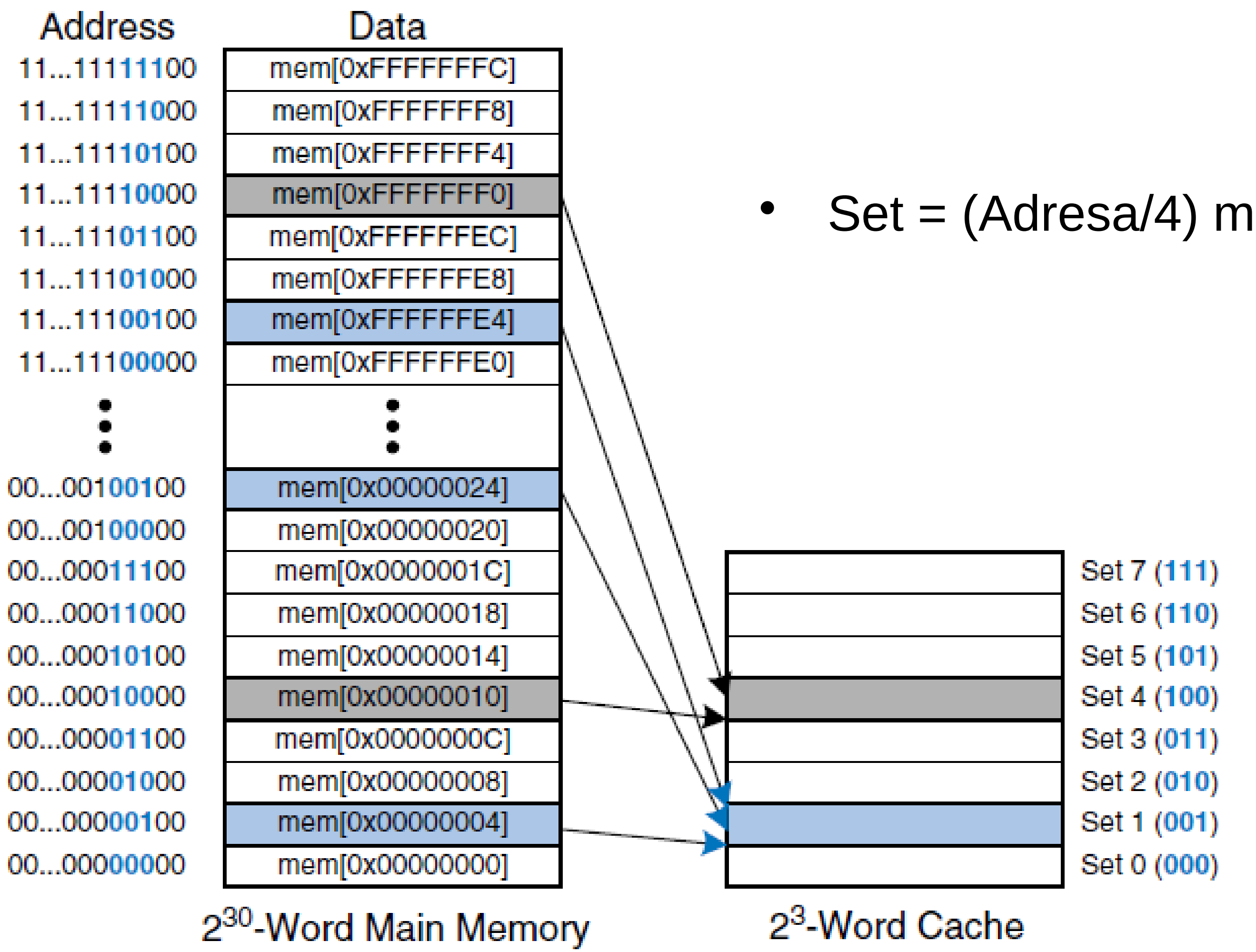


- As for B4M35PAP use more precise definition:
  - Cache block – data, which are transferred to/from memory cache.
  - Cache line includes in addition management informations about block (Tag, valid, dirty, etc. – depends on protocol)
  - Cache row – corresponds to internal cache organization

Example

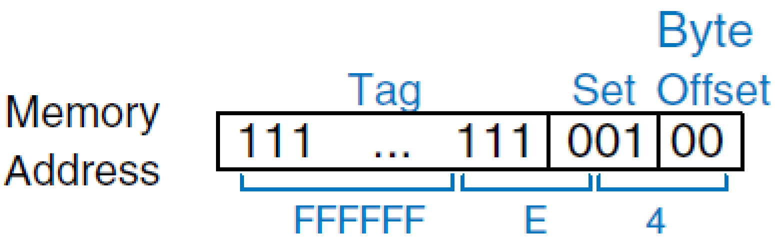
- Consider cache of size 8 one word blocks. Where are stored data/block from address 0xF0000014?
  - Fully associative,
  - Directly mapped, or
  - Or limited number of ways (N=2) associative (2-way cache).

Directly mapped cache



- Set = (Adresa/4) mod 8

Directly mapped cache



Directly mapped cache: one block in each set

Capacity – C

Number of sets – S

Block size – b

Number of blocks – B

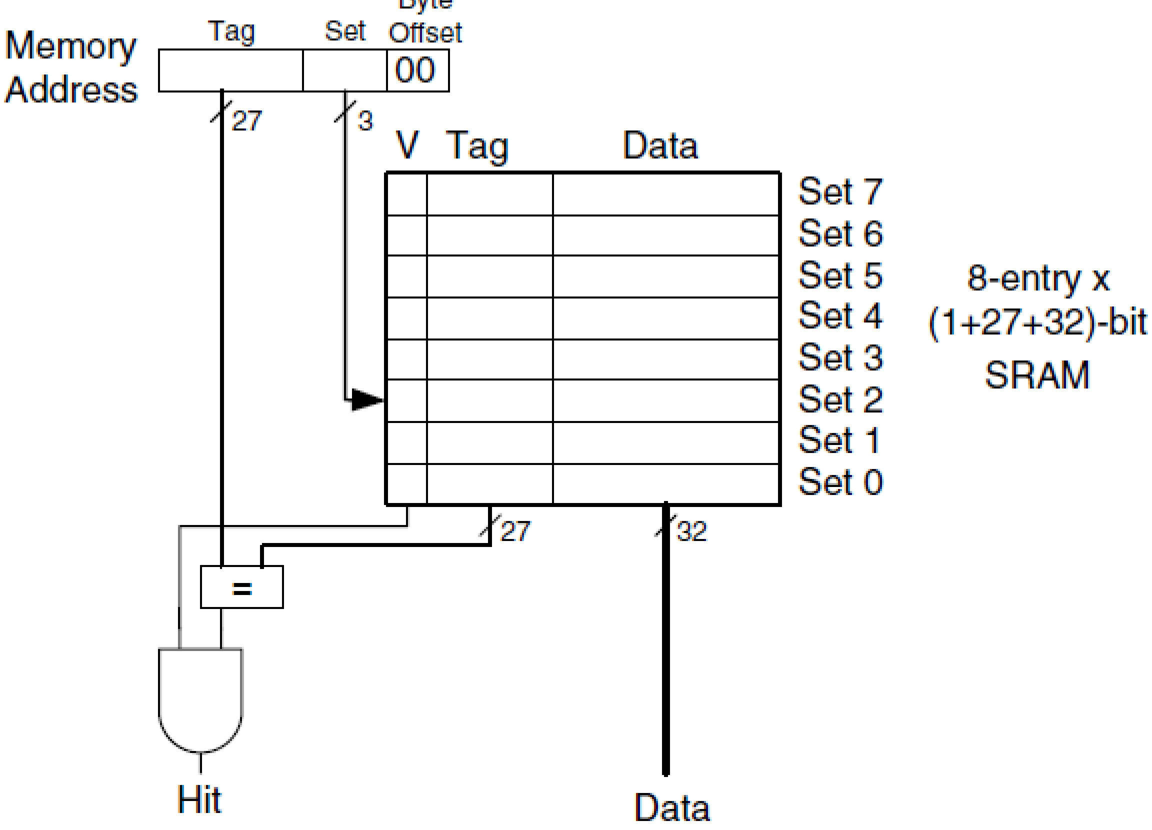
Degree of associativity – N

C = 8 (8 words),

S = B = 8,

b = 1 (one word in the block),

N = 1

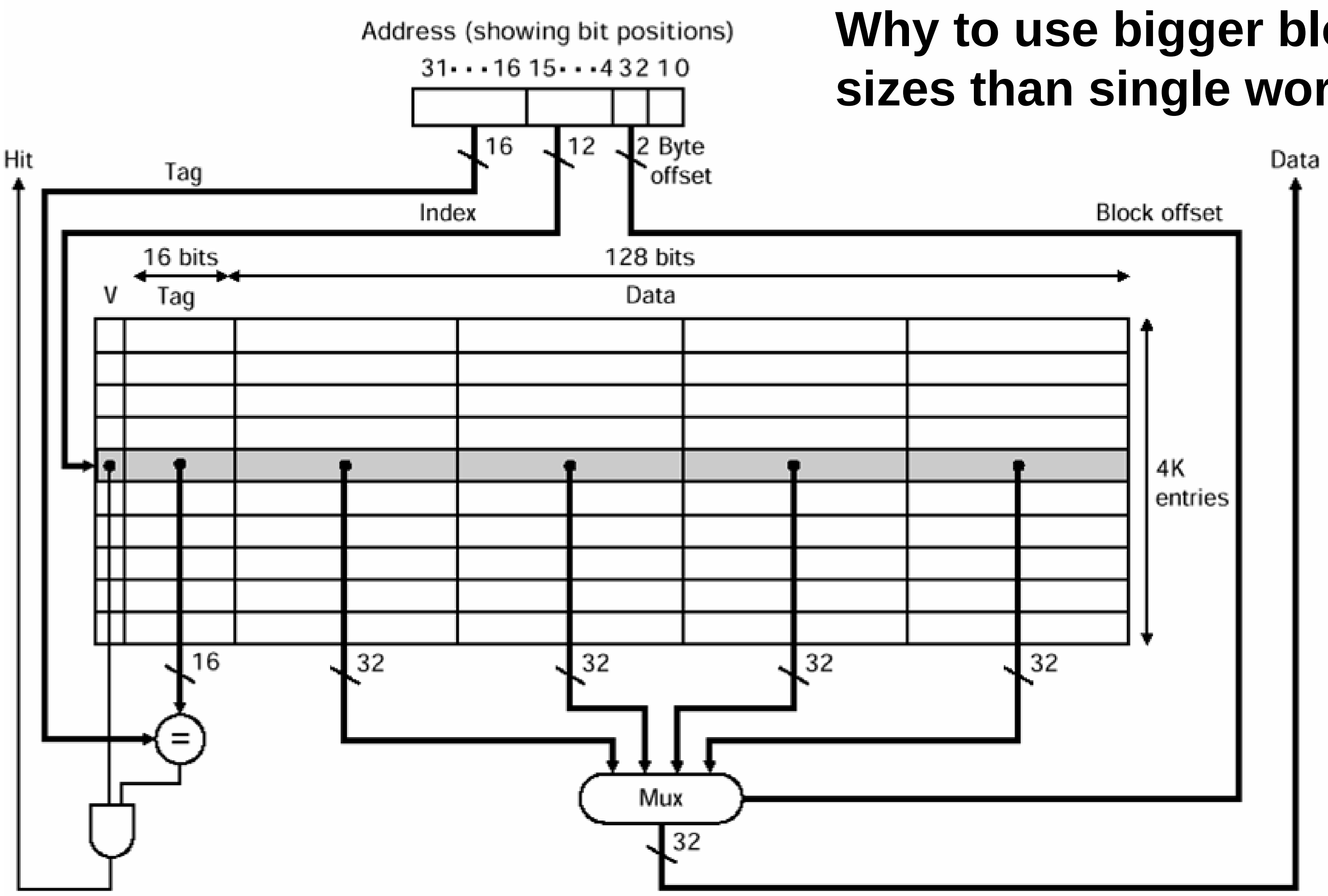


More realistic cache line organization

- **Tag** upper bits of block index in memory (corresponds to memory address divided by size of one way of the cache).
- **Data** field holds actual content/values corresponding to the cached address/addresses.
- **Validity bit** – indicates if Data field is valid, holds real data or is unused/unsynchronized.
- **Dirty bit** – distinguishes state of data field. Informs that value hold in cache (cache) **differs** to value in main memory, that is updated and not write back yet.

V	More flags, i.e. D	Tag	Data

Direct mapped cache implementation – block size 4 words



Why to use bigger block sizes than single word?

Cache with limited degree of associativity – N=2 (ways)

What is advantage of higher associativity degree?

Capacity – C

Number of sets – S

Block size – b

Number of blocks – B

Degree of associativity – N

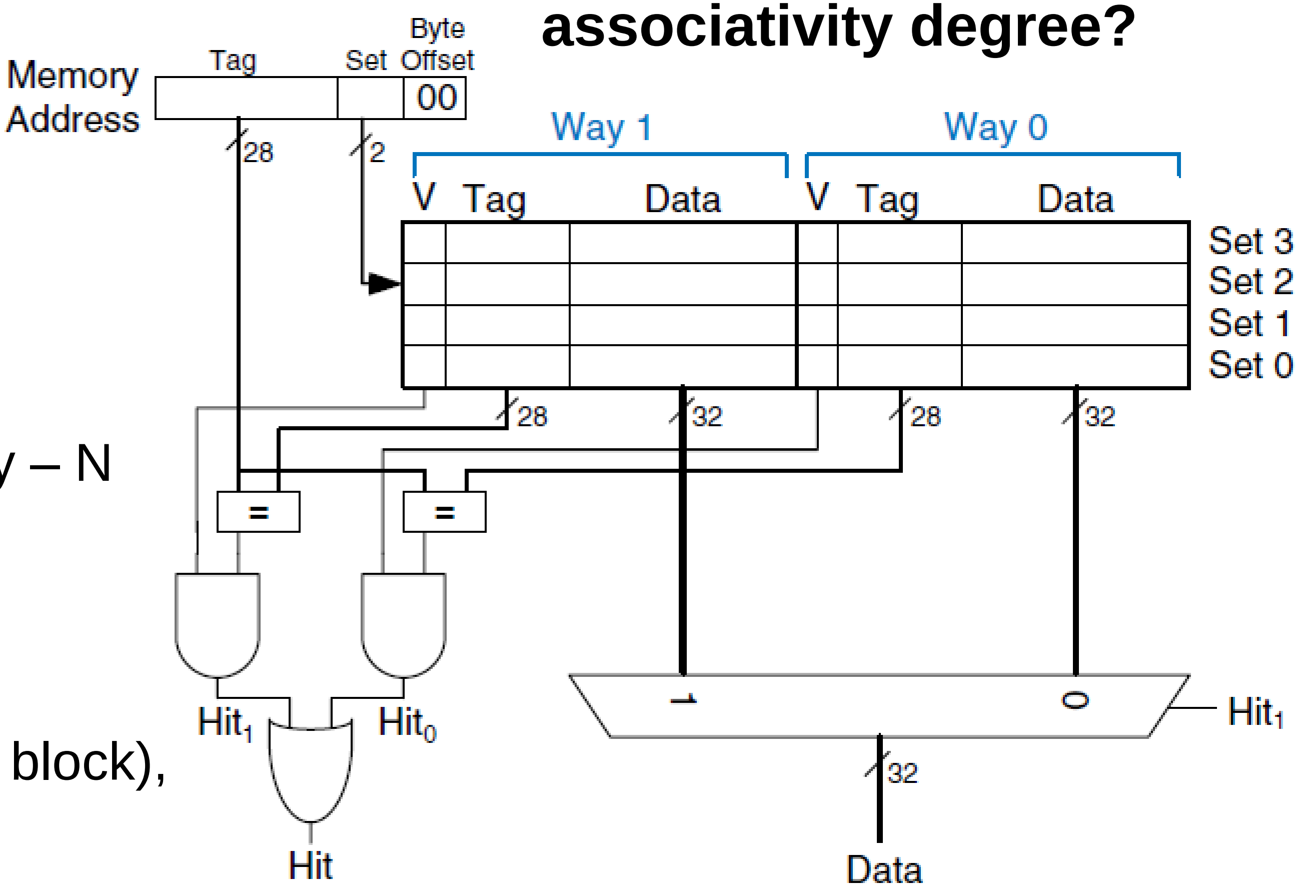
C = 8 (8 words),

S = 4,

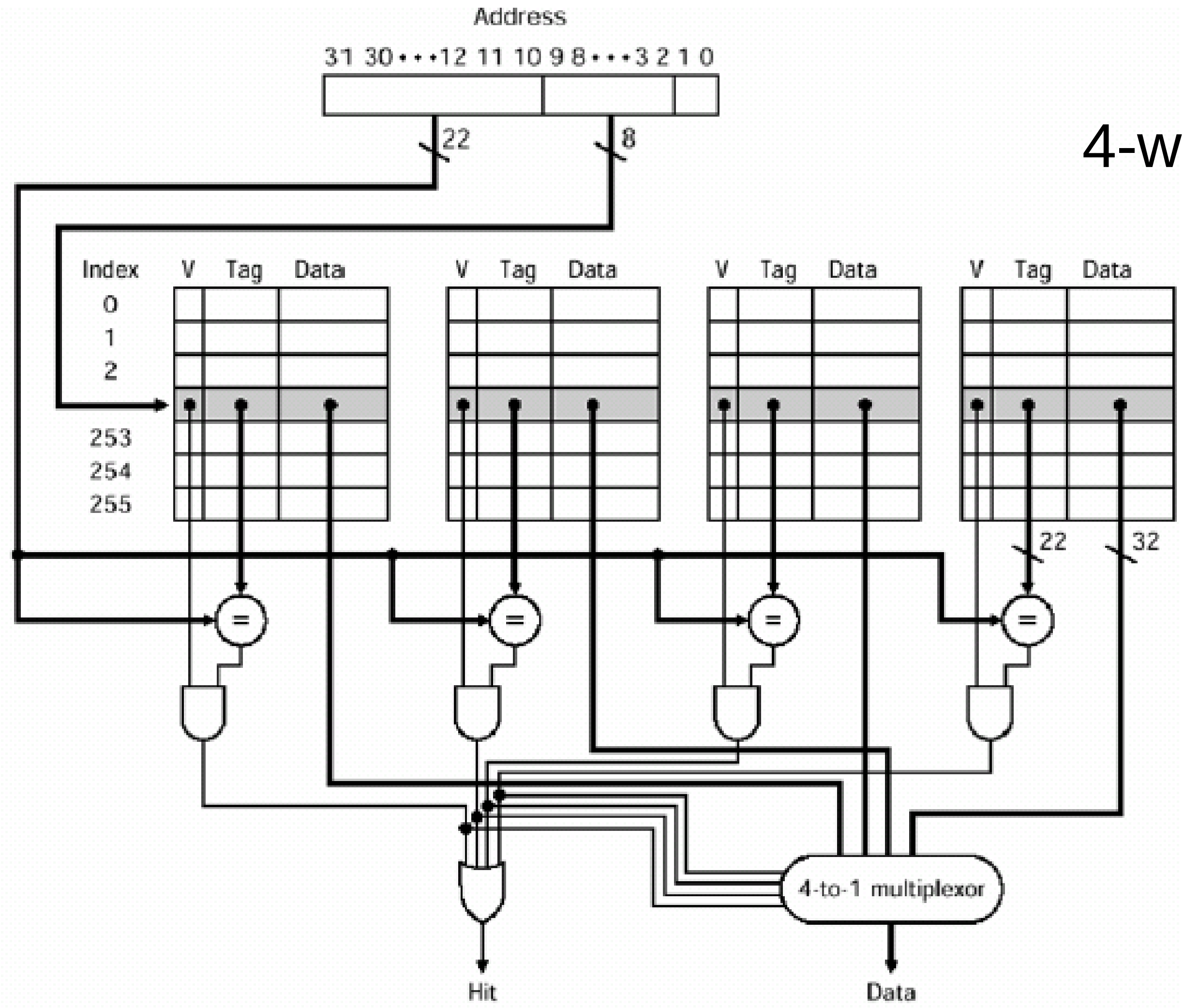
b = 1 (one word in the block),

B = 8

N = 2



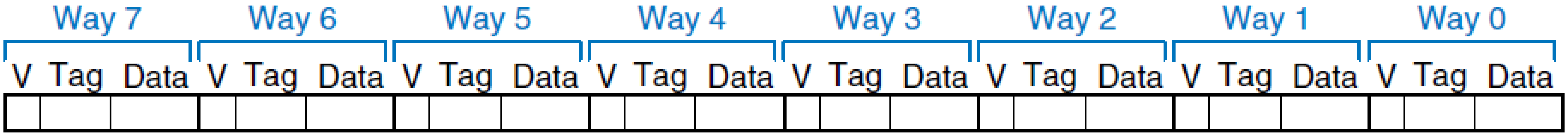
Cache with limited degree of associativity – N=4 (ways)



4-way cache

Fully associative cache

- The fully associative cache contains only one set, the degree of associativity is equal to the number of blocks (N = B). The memory address can be mapped anywhere.
- ... is another naming for a B-way associative cache with one set
- ... has the least amount of conflicts for the given capacity but needs the most HW means (comparators) - the chip surface is growing
- ... is suitable for a relatively small cache sizes



A fully associative cache has only S=1 set.

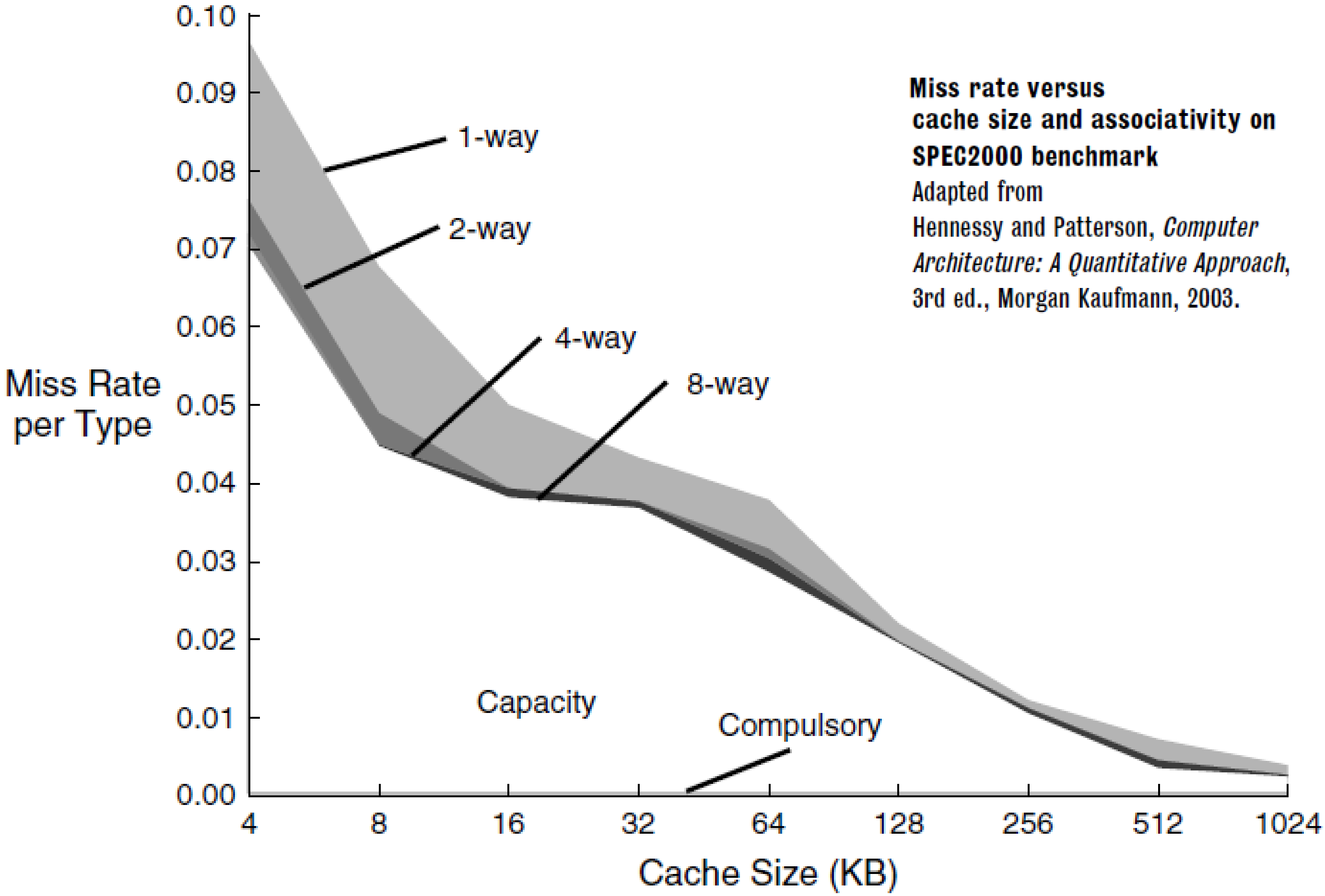
Important cache access statistical parameters

- **Hit Rate** - number of memory accesses satisfied by given level of cache divided by number of all memory accesses
- **Miss Rate** – same principle, but for requests resulting in access to slower memory = 1 – Hit Rate.
- **Miss Penalty** – time required to transfer block (data) from lower/slower memory level.
- **Average Memory Access Time (AMAT)**

  AMAT = HitTime + MissRate×MissPenalty

  Miss Penalty for multilevel cache can be computed by recursive application of AMAT formula

Comparison of different cache sizes and organizations

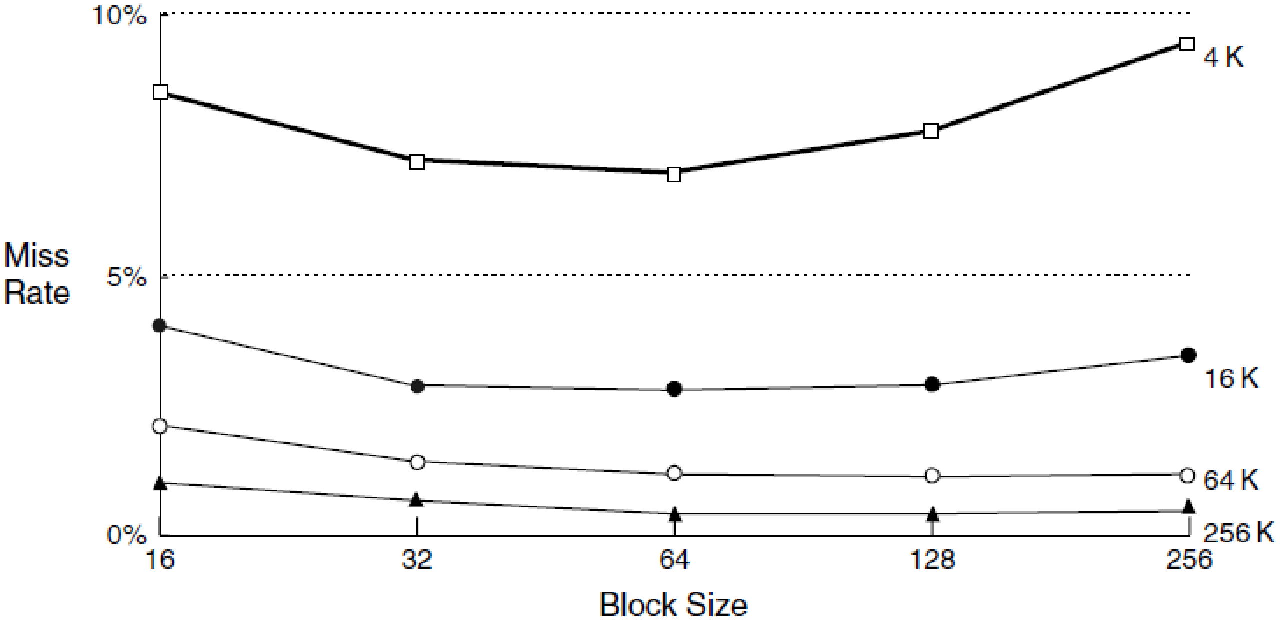


Miss rate versus cache size and associativity on SPEC2000 benchmark
Adapted from
Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, 2003.

Remember: 1. miss rate is not cache parameter/feature!
2. miss rate is not parameter/feature of the program!

What can be gained from spatial locality?

Miss rate of consecutive accesses can be reduced by increasing block size – expected spatial locality. On the other hand, increased block size for same cache capacity results in smaller number of sets and higher probability of conflicts (set number aliases) and then to increase of miss rate. Solution can be combination with prediction of locations for prefetch or prefetch control instructions.



Miss rate versus block size and cache size on SPEC92 benchmark Adapted from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, 2003.

Resolve **Cache Miss** situation, data are not present in cache

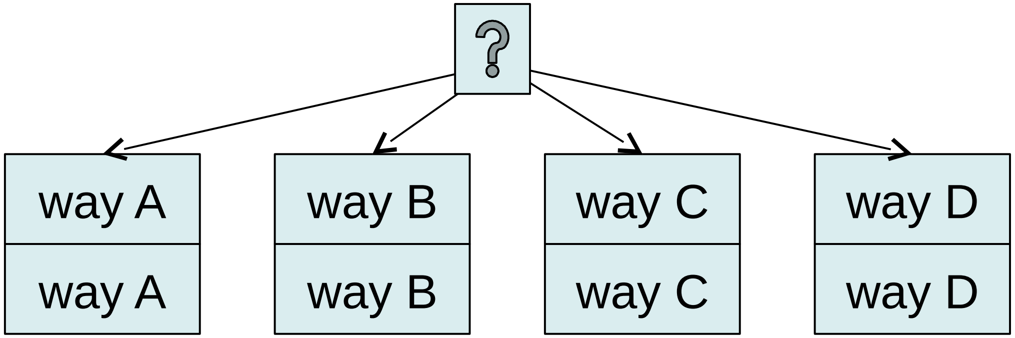
- Data has to be filled from main memory, but quite often all available cache locations which address can be mapped to are allocated

- **Cache content replacement policy** (offending cache line is invalidated either immediately or after data are placed in the write queue/buffer)
- **Random** – random cache line is evicted. Simple but not optimal.
- **LRU** (Least Recently Used) – additional information is required to find cache line that has not been used for the longest time
- **LFU** (Least Frequently Used) – additional information is required to find cache line that is used least frequently – requires some kind of forgetting
- **ARC** (Adaptive Replacement Cache) – combination of LRU and LFU concepts

- **Write-back** – content of the modified (dirty – D bit) cache line is moved to the write queue when line is to be used for another address. Procesed automatically by HW.

Resolve **Cache Miss** situation, data are not present in cache

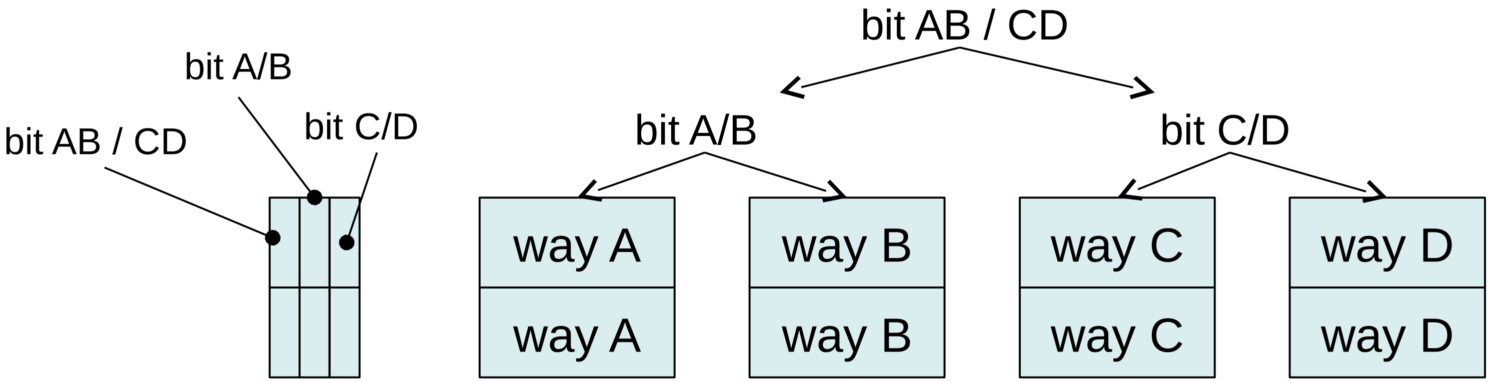
- How is LRU (Least Recently Used) policy implemented ???
- Consider 4-way asociative cache!



- It is not easy if LRU processing is expected to be implemented fast …
- Intel uses pseoudo-LRU pro 4-way, each set has only 3 additional bits (full LRU requires to encode one of 4! = 24 permutations i.e. 5 bits)
- Cache activity is different for two ceases, data are found (cache hit), data re not found (cache miss)
- In the case of a hit, we need to keep track of what way has provided data - certainly <u>not the least used</u> one (i.e. the most recently used)
- In the case of a miss, we have to decide where to save the new data

Resolve **Cache Miss** situation, data are not present in cache

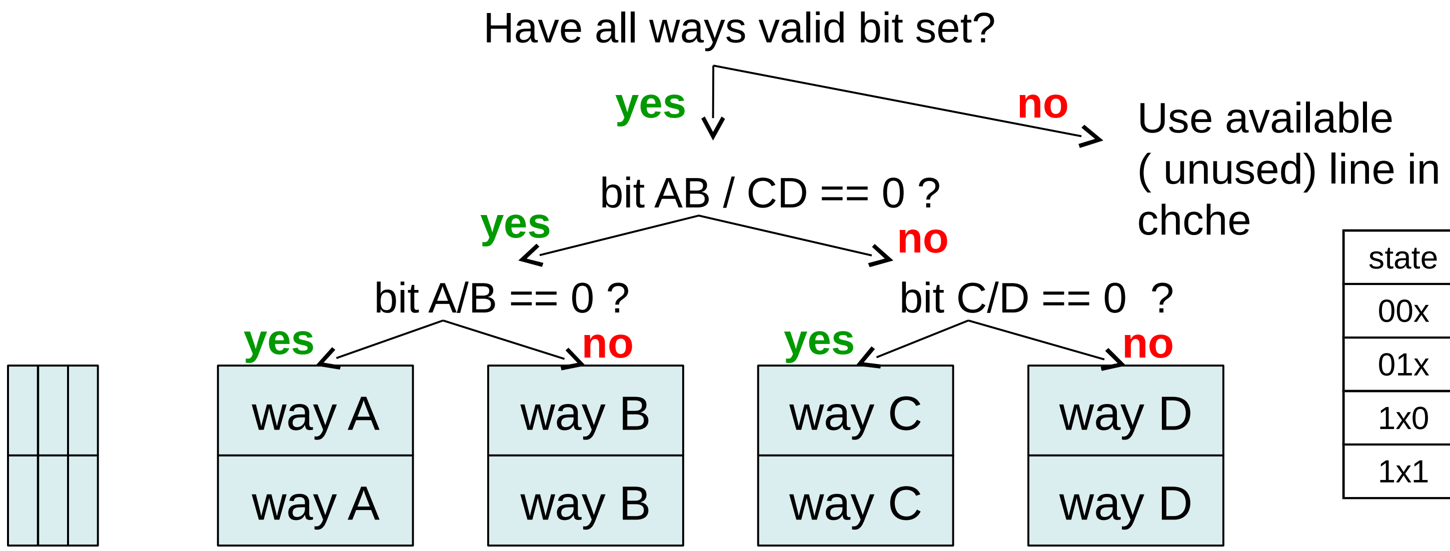
- How is LRU (Least Recently Used) policy implemented ???



- The first bit „AB/CD“ is set if the hit is in A or B. The bit is cleared if ht is in C or D. No information is changed for miss case. This bit „AB/CD“ informs in which “half” was last hit.
- Recurrently, bit „A/B“ is set if hit is in A and cleared for hit in B.
- Similarly for „C/D“ bit.

Resolve **Cache Miss** situation, data are not present in cache

- How is LRU (Least Recently Used) policy implemented ???
- Which data (way) are replaced in cache miss case???



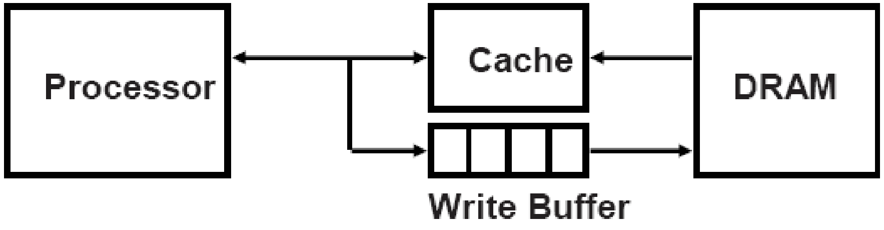
state	replace	next state
00x	way A	11u
01x	way B	10u
1x0	way C	0u1
1x1	way D	0u0

x – don’t care
u - unchanged

- **Sumarize:** Pseudo-LRU advantage is that all bits „AB/CD“, „A/B“, „C/D“ are only written or unchanged in cache hit case. Slowdown (read) occurs only for cache miss case. Utilization of binary tree is easinly extendable for pseudo-LRU implementation for 8, 16, etc. ways.

Resolve **Data write** by processor into memory

- There is cache in between!
- **Data consistency** – logical requirement to ensure same content for given address on all hierarchy levels.
- **Write through** – as data are written into cache they are written to write queue-buffer and then are written asynchronously into memory.
- **Write back** – data are updated only in cache and Dirty (D bit of line metadata) is set. Actual write into memory is initated when given cache-line is to be reused for other content or when sync is required.
- **Dirty bit** – additional bit in cache-line metatada. Marks situation when cache holds **modified** value and main memory requires update.

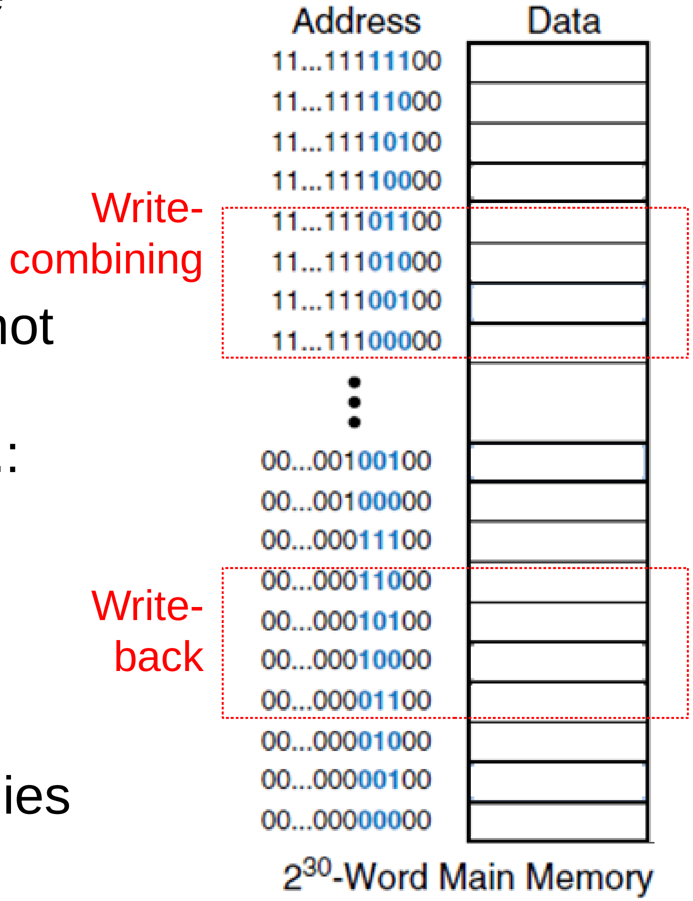


V	Další bity, např. D	Tag	Data

Resolve **Data write** by processor into memory

There are more variants of write strategies:

- **Write-combining** (data are collected in **write combine buffer**. There are written together later; it does not guarantee ordering (weakly ordered memory); example: write to video/framebuffer RAM of graphic card)
- **Uncacheable** (typically when address does not target RAM/main memory => it usually corresponds to write into device registers, i.e.: PCIe card which has BAR mapped to this address
- **Write-protect**
- x86 architecture uses **Memory Type Range Registers** (MTRR) registers for these strategies selection or **Page Attribute Table** (PAT) on newer CPUs which allows per page attributes specification



Trend – **Multiple levels cache**

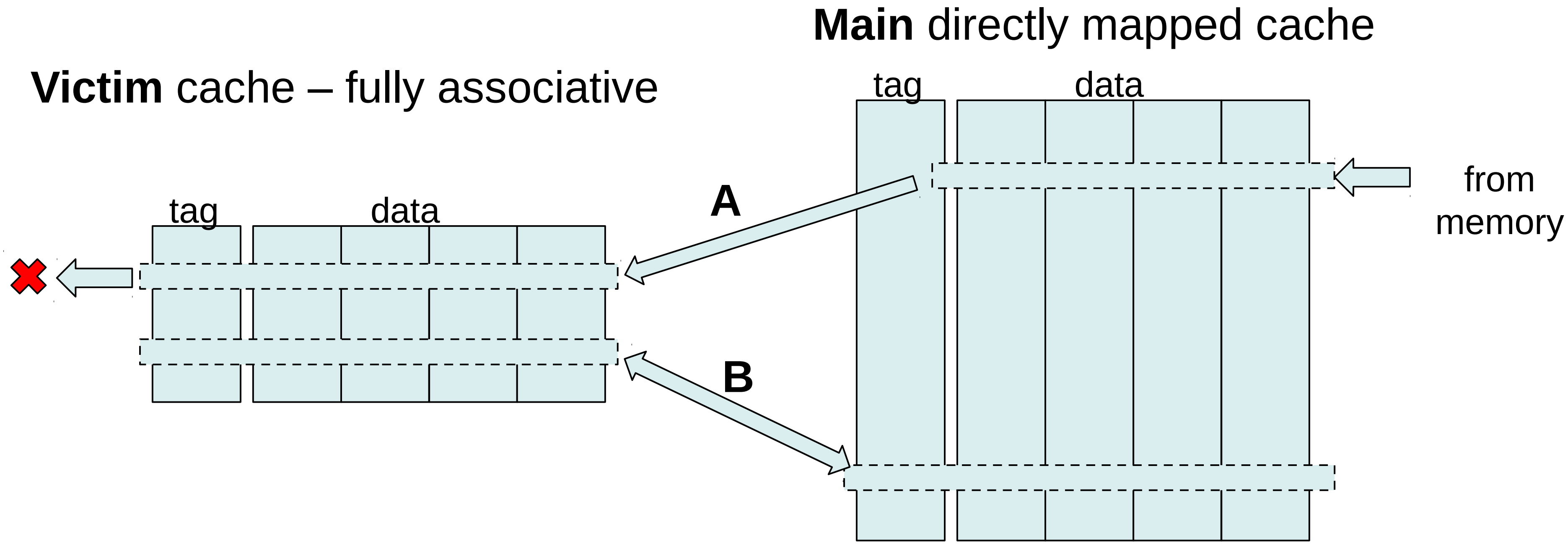
- Primary cache is directly connected to the processor
  - Fast, small. The most important: minimal Hit Time
- L2 Cache resolves misses in primary cache
  - Larger, slower, but still much faster than main memory. Usually shared between cores cluster. The most important: low Miss Rate
- Main memory resolves misses in the last cache level
- Today high performance system use even L3 cache

Parameter	**typical for L1**	**typical for L2**
Počet bloků	250-2000	15 000-250 000
KB	16-64	2 000-3 000
Velikost bloku v B	16-64	64-128
Miss penalty (v hod)	10-25	100-1 000
Miss rates	2-5%	0,1-2%

Victim cache

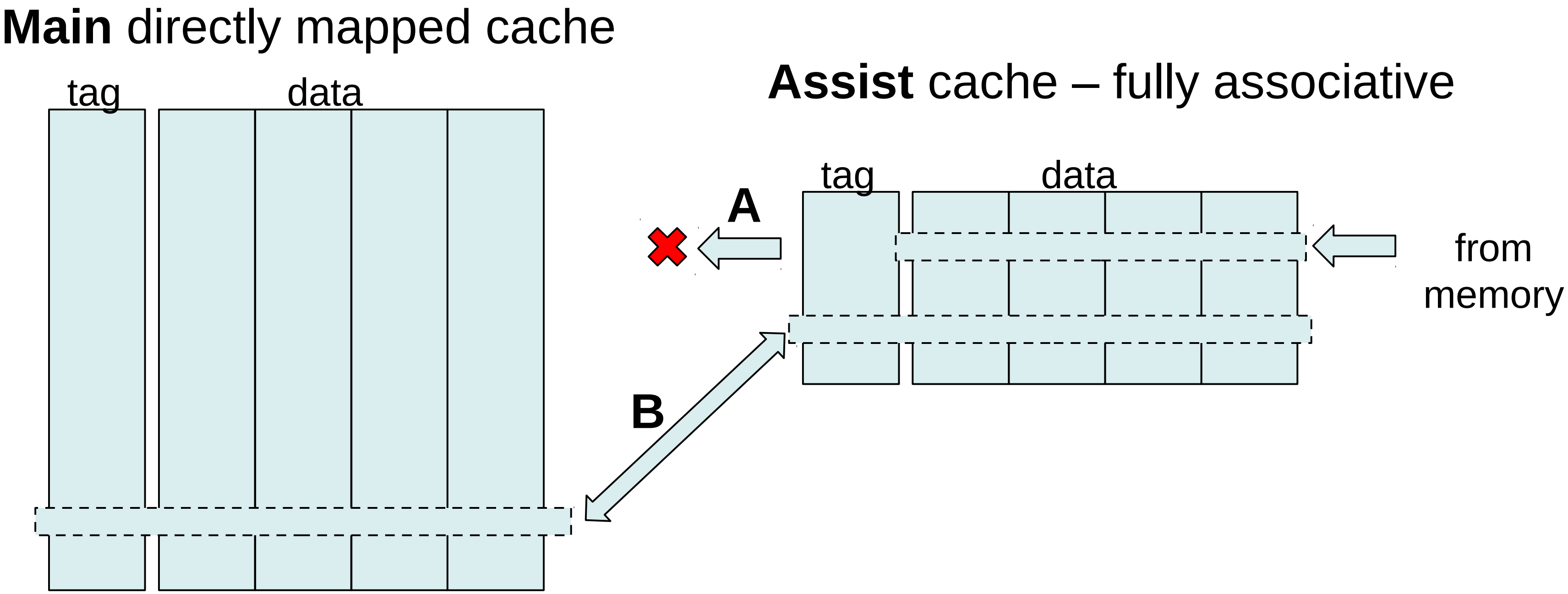
- Directly mapped cache is cheap and fast
- The problem of the directly mapped cache is that in case of a collision (the aliasing mapping of two different addresses) older (often still useful) data/instructions are replaced by newer ones
- Solving this problem was N-way associative, resp. fully associative cache.
- **Is this the only solution? No!** You can still use the so-called Victim cache.
- PRINCIPLE: Use a fast, directly mapped cache. If we remove data from this cache, we would store it in the Victim cache. In cache miss, data are additionally searched in the victim cache before access to the main memory.

Victim cache


Main directly mapped cache
Victim cache – fully associative
tag
data
tag
data
A
B
from memory

- **A**: The incomming block is placed into main cache and „expelled“ block into Victim cache (FIFO strategy for Victim cache is sufficient to realize LRU – result of B rule)
- **B**: In the case of miss in main cache and hit in Victim cache, cache lines are swapped between these caches
- **Is this only alternative? No!** Assist cache is next alternative.

Assist cache


Main directly mapped cache
tag
data
Assist cache – fully associative
tag
data
A
B
from memory

- **A**: Incoming block is stored into Assist cache (FIFO)
- **B**: If there is miss in main cache and hit in Assist cache, swap cache lines between caches.
- Remarks: Data are transferred into main cache only after hit in Assist cache, that is, after repeated requests to access the same address. Therefore, the data cached in main cache prove time locality.

Do you understand to this lecture?

- If so, you are already aware that using 2 principles (temporal and spatial localityprinciples) can lead to a significant speedup of your program by using cache effectively ... !!!
- There are HW and SW (compiler) techniques that optimize caching based on these principles. You can not influence HW techniques from a programmer's point of view. You can set optimization level for compiler ...
- However, even the best compiler only compiles what the programmer wrote. Algorithm selection, storage and manipulation of data structures are all determined by the programmer. Therefore, "the most" work is still in the hands of the programmer, and it depends to a large extent on programmer how "fast" program will be .

Do you understand to this lecture?

- Instruction cache – more complex
  - Appropriate code reordering, eventually reordering/grouping of hot or interconnected functions in memory
  - Profiling
- Data cache – easier
  - Proper data layout – the data we plan to use sequentially, sort sequentially in memory, etc.
  - Merge fields or related data structures, locate the most used fields first in structures
  - Work on data blocks - use the already used one as soon as possible
  - Iteration in nested cycles - see introductory example - to browse the memory sequentially and not with skips
  - merging two loops into one - Loop fusion, etc.

Do you understand to this lecture?

- Spatial locality – conflicts/aliasing in cache:

```
/* Before optimization */
int values[SIZE];
int keys[SIZE];
int scores[SIZE];

/* After optimization */
struct item{
 int value;
 int key;
 int score;
};
struct item records[SIZE];
```

Assume 2-way associative cache…

```
for(i=0; i<SIZE; i++)
  for(j=0; j<SIZE; j++)
    …
```

Do you understand to this lecture?

- Temporal locality:

```
/* Před optimalizací */
for (i = 0; i < SIZE; i++)
   for (j = 0; j < SIZE; j++)
      a[i][j] = b[i][j] * c[i][j];
for (i = 0; i < SIZE; i++)
   for (j = 0; j < SIZE; j++)
      d[i][j] = a[i][j] - c[i][j];

/* Po optimalizaci */
for (i = 0; i < SIZE; i++)
   for (j = 0; j < SIZE; j++)
      { a[i][j] = b[i][j] * c[i][j];
      d[i][j] = a[i][j] - c[i][j];}
```

It's not just about saving the instructions, but also using the cache more efficiently ...

Do you understand to this lecture?

- Next example – matrix multiplication

```
for(i=0; i < N; i++)
  for(j=0; j < N; j++) {
    tmp = 0;
    for (k=0; k < N; k++)
       tmp += y[i][k]*z[k][j];
    x[i][j] = tmp;
  }
```

Will it help us somehow when swap these two lines? Will the program be equivalent?

(See introductory example ...)

Do you understand to this lecture?

- Next example – matrix multiplication
- It is better to use so-called block multiplication.
- Idea: Let's divide the calculation into BxB sub-matrices that will fit in cache ... => elimination of "capacity misses"

```
for (jj = 0; jj < N; jj = jj+B)
  for (kk = 0; kk < N; kk = kk+B)
    for (i = 0; i < N; i++)
      for (j = jj; j < min(jj+B-1,N); j++) {
      tmp = 0;
        for (k = kk; k < min(kk+B-1,N); k++)
           tmp += y[i][k]*z[k][j];
        x[i][j] = x[i][j] + tmp;
      }
```

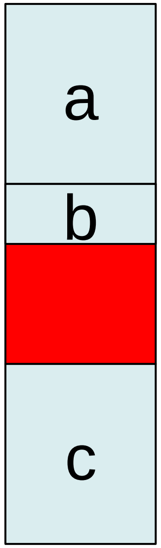
More to read: http://suif.stanford.edu/papers/lam-asplos91.pdf

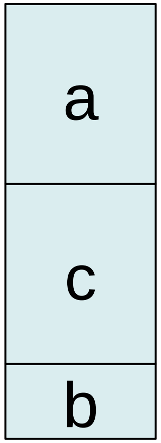
Do you understand to this lecture?

- Do not waste the memory – use minimal required amount of memory
- Do you see differences in these declarations?

-
  ```
  /* Before optimization */
   int a=0;
   char b='a';
   int c=1;
  ```

-
  ```
  /* After optimization */
   int a=0;
   int c=1;
   char b='a';
  ```





What is incorrect? – see. pahole

```
struct cheese {
  char name[17]; /* 0 17 */
     /* XXX 1 byte hole, try to pack */
  short age; /* 18 2 */
  char type; /* 20 1 */
     /* XXX 3 bytes hole, try to pack */
  int calories; /* 24 4 */
  short price; /* 28 2 */
     /* XXX 2 bytes hole, try to pack */
  int barcode[4]; /* 32 16 */
};   /* size: 48, cachelines: 1 */
     /* sum members: 42, holes: 3 */
     /* sum holes: 6 */
     /* last cacheline: 48 bytes */
```

Arnaldo Carvalho de Melo: The 7 dwarves: debugging information beyond gdb

Lessons learned

- Be careful about the layout of the structure members
- Place the most critical elements (most commonly used) ones at the beginning of the structure
- If you access structure members, try to keep the order in which they are defined in the structure

- For larger structures, the rules also apply and can be applied for the cache line size

- The other question is what members should be in the structure at all: OOP principle vs. speed

Do you understand to this lecture?

- **Data, which are accessed in same time instant (sequentially) group together.**
- **Data, which are often accessed, group together.**
- Data alignment in memory has to be often analyzed as well – directly in assembly language or in C – check if your compiller aligns allocated memory to 8-byte border for doubles, if not:
  - Allocate as much as you need + 4B (or even more – according to data size)
  - use AND to obtain aligned store for your data, example:

```
double a[5];
double *p, *newp;
p = (double*)malloc ((sizeof(double)*5)+4);
newp = (double*)((intptr_t)(p+4)) & (-7);
```

- See also int posix_memalign(void **memptr, size_t align, size_t size);

Do you understand to this lecture?

- Prime numbers search – Sieve of Eratosthenes:

```
/* Before optimization */
 boolean array[max];
 for(i=2;i<max;i++) {
   array = 1;
 }
 for(i=2;i<max;i++)
   if(array[i])
     for(j=i;j<max;j+=i)
          array[j] = 0; /* transfer from memory to cache
                           write 0*/
```

Transfer occurs only for cache miss

Do you understand to this lecture?

- Prime numbers search – Sieve of Eratosthenes:

```
/* After optimization */
boolean array[max];
for(i=2;i<max;i++) {
  array = 1;
}
for(i=2;i<max;i++)
  if(array[i])
    for(j=2;j<max;j+=i)
       if(array[j]!=0) /* transfer from memory into cache
                        and read */
          array[j] = 0; /* write 0 only if required */
```

- It reduces useless writes (reduces writes to main memory – dirty cache lines has to be written before line reuse)

Cache bypass can speed up your programs for some cases

- If you are producing data, which are not used in short time (*non-temporal* write operation), there is no reason to cache it
- This is often the case for large data structures (matrices, etc.)
- Why does this speed up the program?

```
#include <emmintrin.h>
void _mm_stream_si32(int *p, int a);      And more…
```

It stores data from „a“ variable to „p“ address without forcing caching of location. However, if the "p" already exists in the cache, the cache will be updated.

-> see **Write-combining** strategy;

-> final WC buffer flushing is under programmer control, else by HW

- More details: “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Optimize often called functions

- If you frequently and especially in a fast sequence call for the same function, optimize it! **Use caching principle for that sometimes, be careful about threads …**
- Example: We know that we will need to calculate square roots of integer only and even often only from 0 to 10.

```
double sqrt10(int i) {
    static const double lookup_table[] = {0, 1,
     sqrt(2), sqrt(3), 2, sqrt(5), sqrt(6),
  sqrt(7), sqrt(8), 3, sqrt(10)   };

    if(0 <= i && i <= 10)
     return lookup_table[i];
    else
        return sqrt(i);
}
```

Optimize often called functions

- Example: We will call a function which is called often in succession with the same parameters…

```
double f(double x, double y) {
     return sqrt(x * sin(x) + y * cos(y)); }
```

After optimization, be careful about threads:

```
double f(double x, double y) {
    static double prev_x = 0, prev_y = 0, result = 0;

      if (x == prev_x && y == prev_y)
        return result;
      prev_x = x;
    prev_y = y;
    result = sqrt(x * sin(x) + y * cos(y));
    return result;
}
```

How to determine cache parameters?

- Linux

```
#include <unistd.h>
long sysconf (int name);
```

Kde name:

_SC_LEVEL1_ICACHE_SIZE
_SC_LEVEL1_ICACHE_ASSOC
_SC_LEVEL1_ICACHE_LINESIZE etc.

- Windows

GetLogicalProcessorInformation() -> SYSTEM_LOGICAL_PROCESSOR_INFORMATION which contains CACHE_DESCRIPTOR field

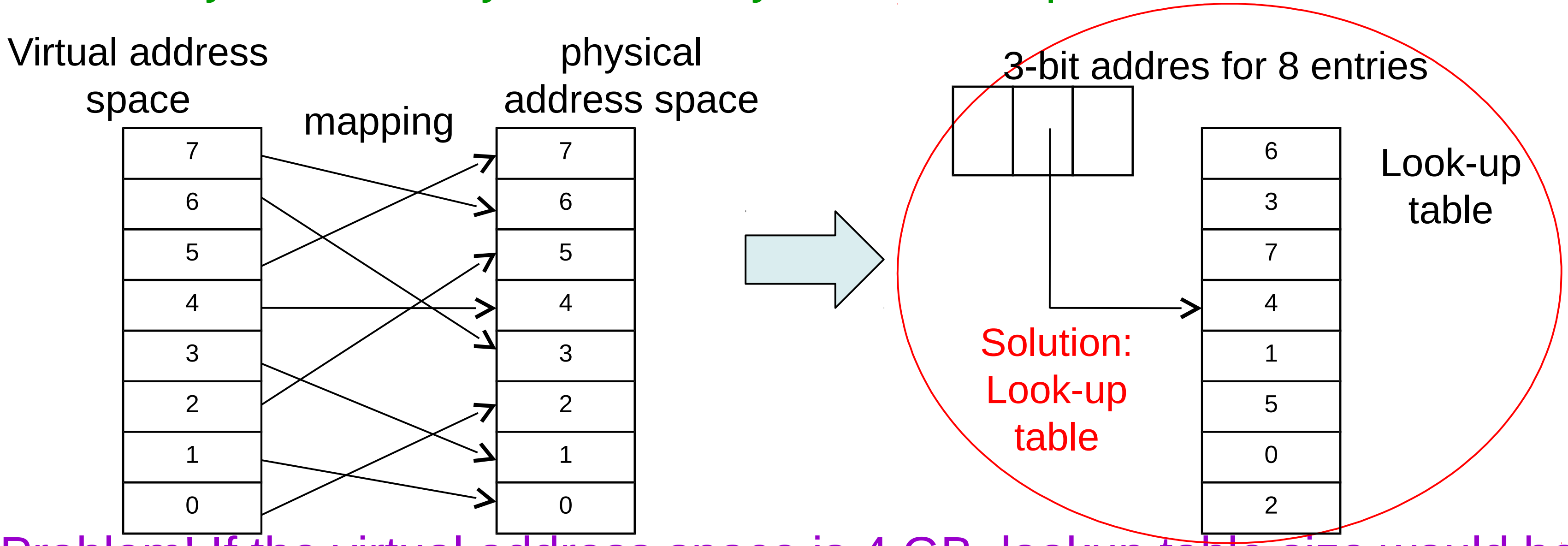
Virtual memory.

Reasons to introduce virtual memory..

- **Many (>10, server >10000) processes run in parallel on a computer**
- **Problem is how to divide and manage physical memory (i.e. 1 GB) between these processes? If single continuous block is provided, required amount is not known in advance? Other problem is corruption of memory by maliscuous program (i.e. virus) or due to error in program (unintended programmer mistake – bad pointers manipulation) which can target block allocated to other process.**
- **Address translation together with virtual memory is solution…**
- **Each process is given the illusion that it has separate memory/ address space allocated to it and can use all pointers values (excluding some specific areas, i.e. range above 3 GB for 32-bit x86).**
- **It is even possible to maintain illusion that each process has whole or even more memory available than is total physical main memory in a system because secondary memory can provide additional space.**
- **Basic idea: Process addresses memory by virtual addresses (own address space) and these addresses are translated to physical ones..**

Reasons to introduce virtual memory..

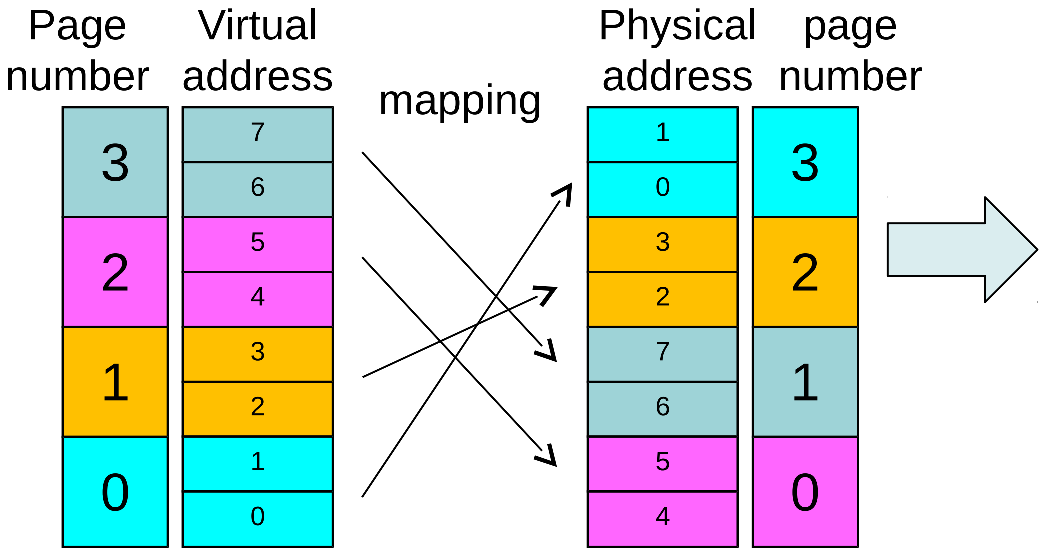
- Explain idea on 8B (Bytes) virtual address space and 8B physical memory
- **Now to implement address translation? Byte addressing expected**.
- **One of the solutions:** Translation of random virtual address to random physical address is required. I.e. 3-bit virtual address should be translated to 3-bit physical address. It is enough to use table with 8 entries, where each entry holds 3 bity, dohromady 8x3=24bitů/proces.

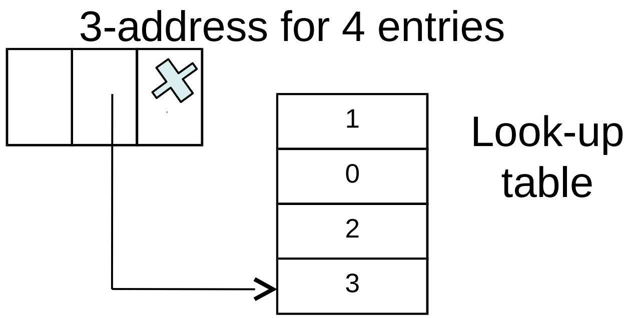


- Problem! If the virtual address space is 4 GB, lookup table size would be 2^{32}x32 bitů = 16GB for each process. This is too much…

Virtual memory - Solve too large table from previous slide:

- **Mappin of each arbitrarily (cell/byte) virtual address to arbitrarily virtual address is practicaly infeasible!**
- **Solution:** Divide virtual address space into blocks of same size – virtual pages, and physical memory on physical pages of same size. In our example, we have a 2B page.





Solution – one bit of address is not used for translation. Look-up table has half of entries which require less bits.

- Our solution then translates virtual addresses on groups basis... Inside the given page some bits define byte offset and are not used during translation. We are thus able to use/map the entire address space.

Example No 1

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int a, b[4], *c, d;
    c = (int*)malloc(4*sizeof(int));
    printf("%p %p %p %p\n",&a,&b,&c,&d);
    printf("%p %p %p\n",&b[0],&b[1],&b[2]);
    printf("%p %p %p\n",&c[0],&c[1],&c[2]);
    free(c);
    return 0;
}
```

What are the consequences?

- Array data are stored sequentially.

More questions ...

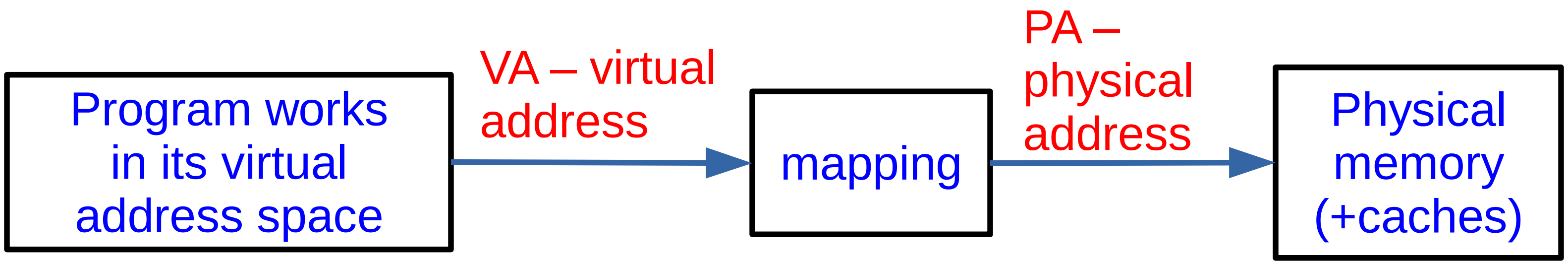
- Which address is it?
- Where are these data mapped in chache and phys. mem?

Program output:

```
0028FF1C 0028FF0C 0028FF08 0028FF04
0028FF0C 0028FF10 0028FF14
00801850 00801854 00801858
```

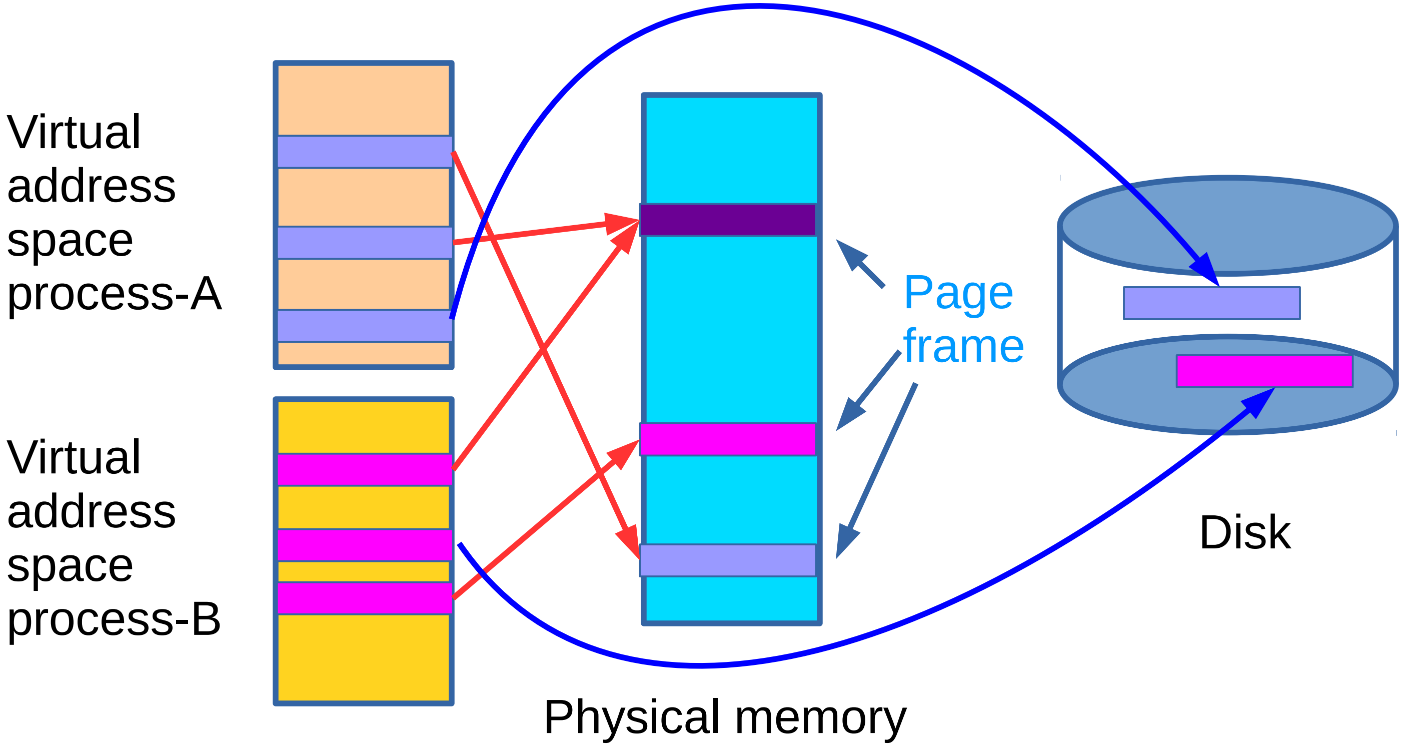
Virtual address and virtual memory

- **Virtual memory (VM)** – a way to manage memory where a separate address space is provided to each process, it is (can be) organized independently on the physical memory ranges and can be even bigger than the whole physical memory
- Programs/instructions running on the CPU operate with data only through virtual addresses
- Translation from virtual address (**VA**) to physical address (**PA**) is implemented in HW (MMU, TLB) fully or can require TLB fill by OS.
- Common OSes implement virtual memory through paging which extends concept even to swapping memory content onto secondary storage (disc)


Program works in its virtual address space
VA – virtual address
mapping
PA – physical address
Physical memory (+caches)

Virtual memory - paging

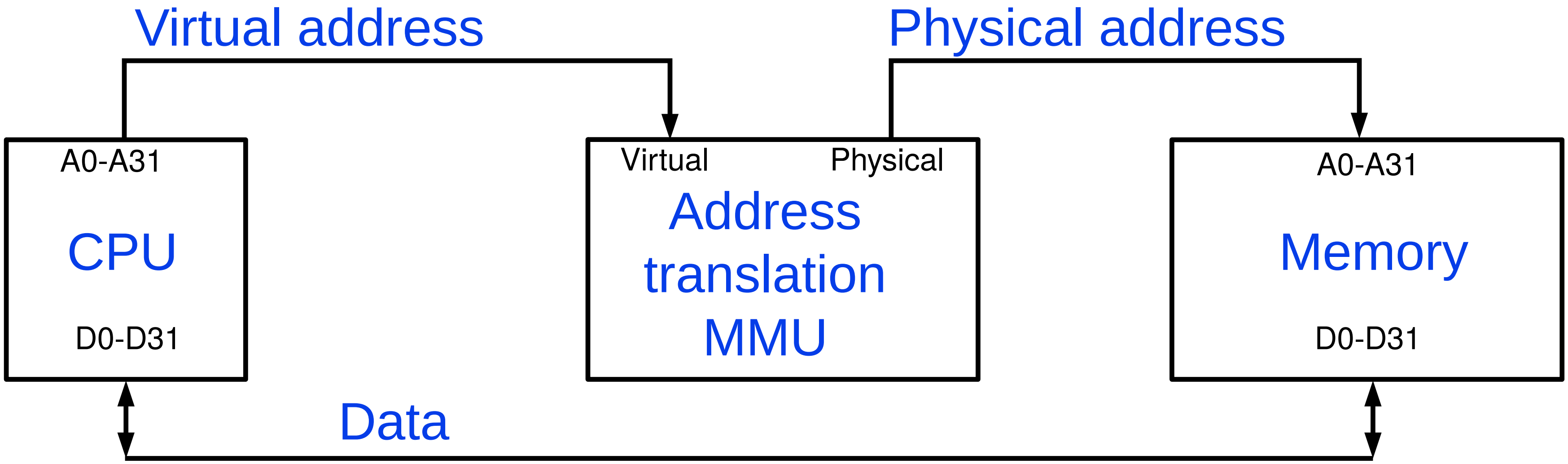
- Process virtual memory content is divided into aligned pages of same size (power of 2, usually 4 or 8 kB)
- Physical memory consists of page frames of the same size
- Note: huge pages option on modern OS and HW – 2^n pages


Virtual address space process-A
Virtual address space process-B
Page frame
Disk
Physical memory

Virtual memory - paging

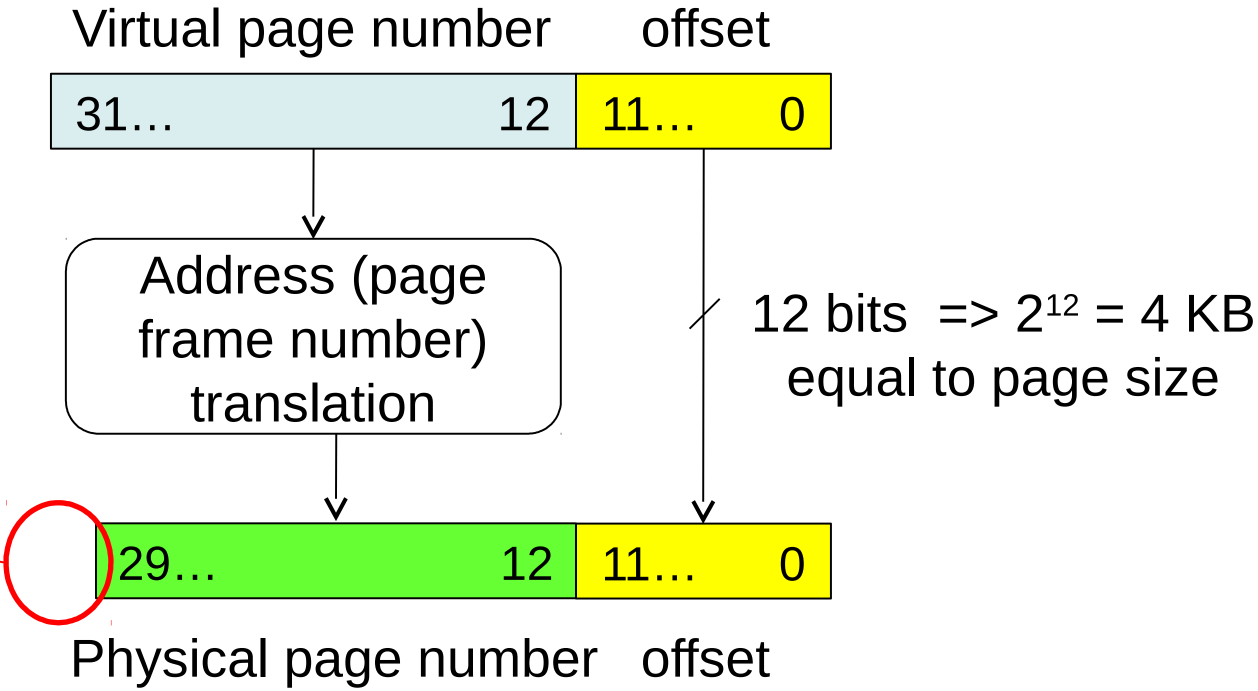
- Each virtual page may map to at most one physical page (vice versa rule is not required)
- Multiple virtual pages may be mapped to one particular physical page. What does it bring?
- We can share memory across different processes or threads (data or code - the OS loads the shared libraries only once), we can provide other privileges (access rights).
- If the program tries to access the page in a way that does not match its permission, the CPU generates a *General Protection fault (SIGSEGV)*
- Handler for General protection fault - a typical reaction is the end of the process

Virtual/physical address and data



Virtual and physical addressing in more detail

- Consider virtual ddress width 32 bits, physical memory size 1GB and 4 KB page size


Virtual page number
offset
31…
12
11…
0
Address (page frame number) translation
12 bits => 2^{12} = 4 KB equal to page size
29…
12
11…
0
Physical page number
offset

What about remaining bits? Described Later …

- What is **very important** practical consequence of this arrangement → the least significant address bits (offset) are unchanged by translation.

Return to example No 1

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int a, b[4], *c, d;
    c = (int*)malloc(4*sizeof(int));
    printf("%p %p %p %p\n",&a,&b,&c,&d);
    printf("%p %p %p\n",&b[0],&b[1],&b[2]);
    printf("%p %p %p\n",&c[0],&c[1],&c[2]);
    free(c);
    return 0;
}
```

Program output:

```
0028FF1C 0028FF0C 0028FF08 0028FF04
0028FF0C 0028FF10 0028FF14
00801850 00801854 00801858
```

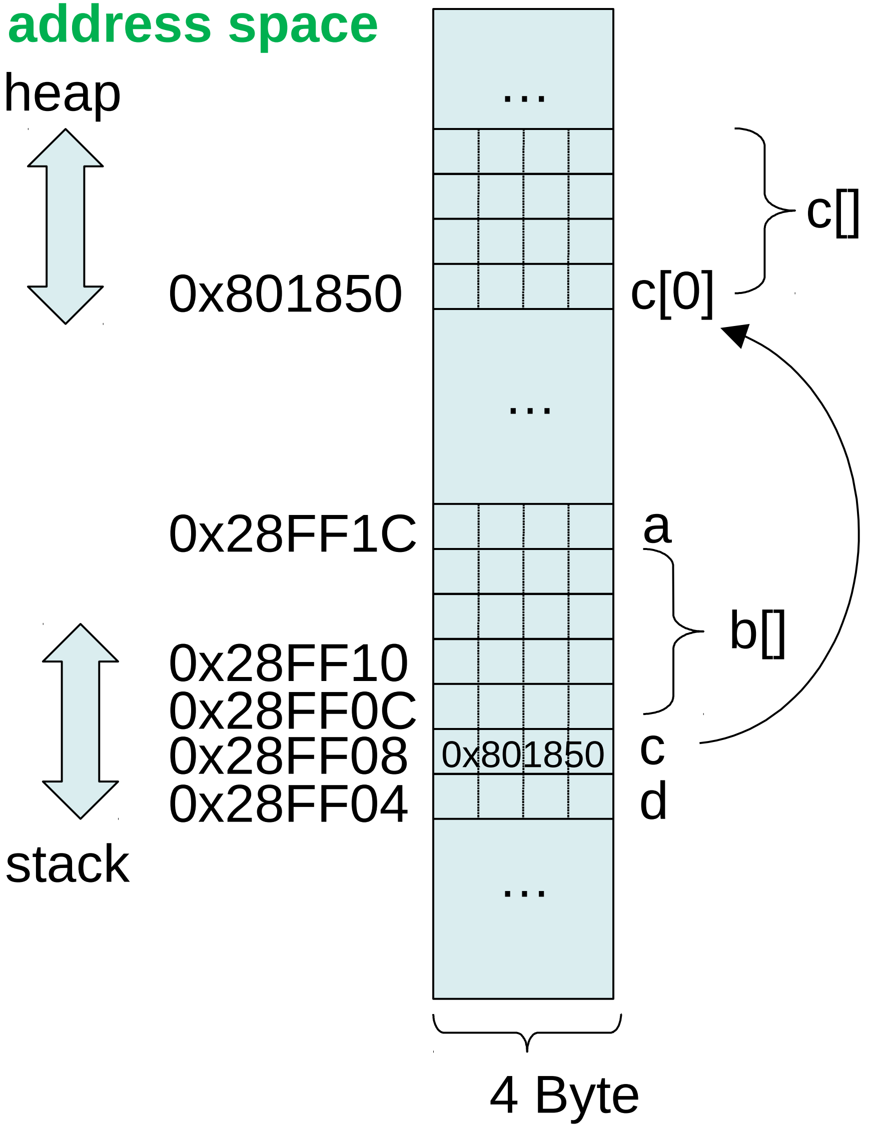
Return to example No 1

- Have you noticed addresses, where ***a*, c, *d*** variables and array ***b*** are located?
- What does it mean if program is extended by commands:

```
a = 1;
b[0] = a+1;
b[1] = b[0]+1;
d = b[2];
```

//b[2] is not initialized..

Virtual address space



Return to example No 1

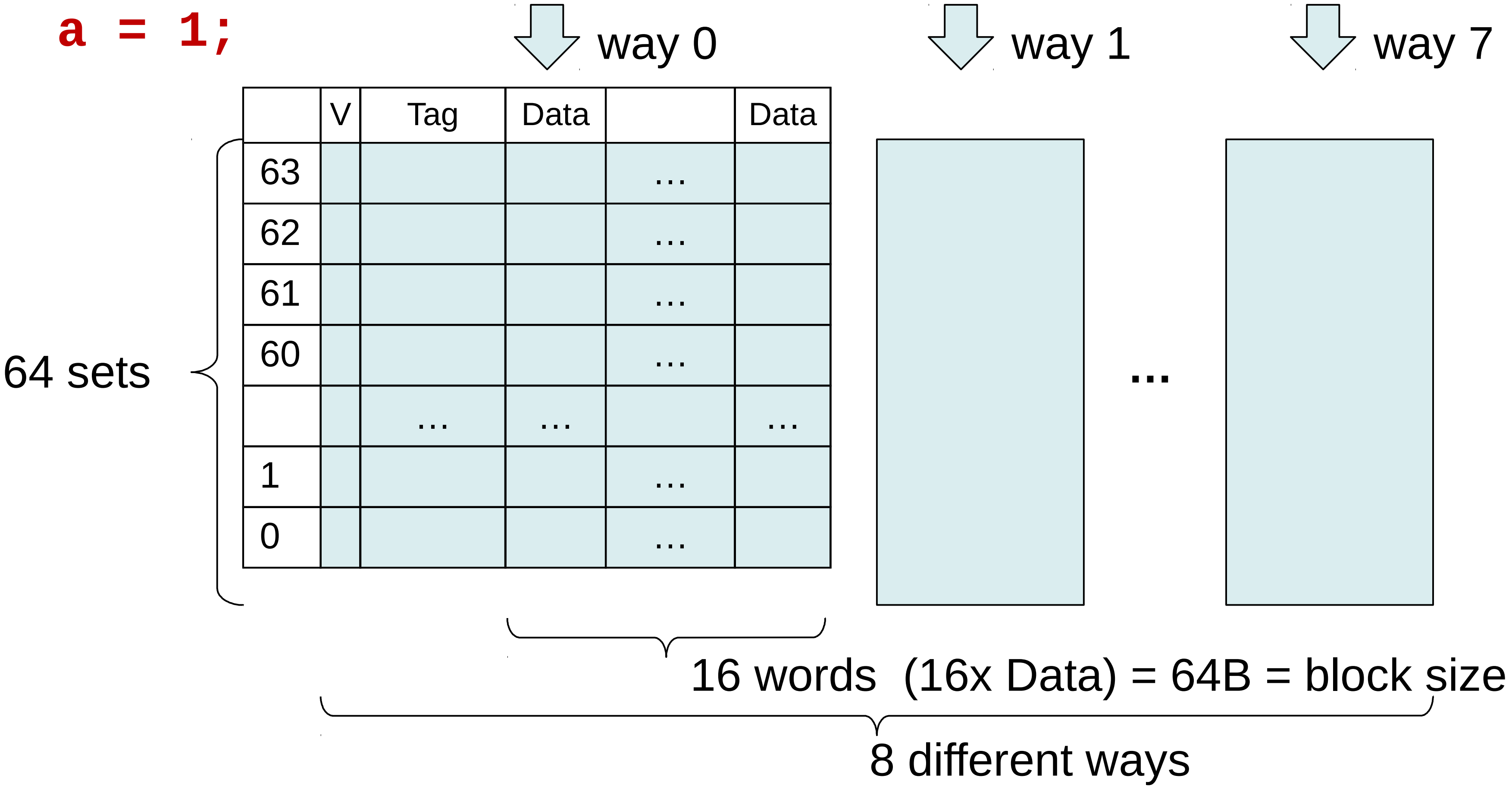
- Consider L1 data cache of size 32kB with associativity degree 8 and block size 64B. Cache is initially empty.
- What happens when the first line of the program is executed?

```
a = 1;
b[0] = a+1;
b[1] = b[0]+1;
d = b[2];
```

Return to example No 1

- Consider L1 data cache of size 32kB with associativity degree 8 and block size 64B. Cache is initially empty.
- What happens when the first line of the program is executed?

```
a = 1;
```

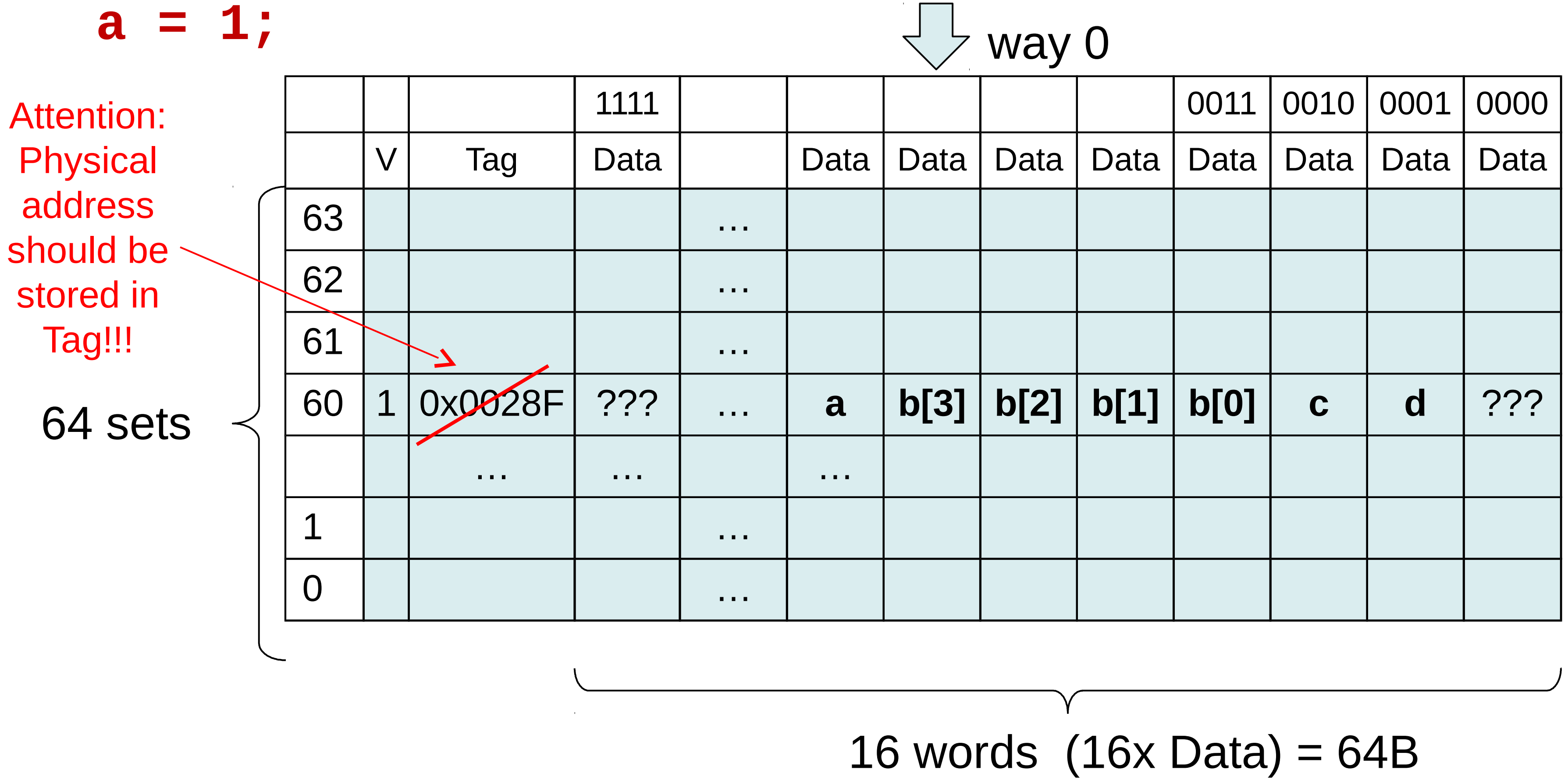


	V	Tag	Data		Data
63				…	
62				…	
61				…	
60				…	
		…	…		…
1				…	
0				…	

Return to example No 1

- Consider L1 data cache of size 32kB with associativity degree 8 and block size 64B. Cache is initially empty.
- What happens when the first line of the program is executed?

```
a = 1;
```



way 0

Attention: Physical address should be stored in Tag!!!

64 sets

			1111						0011	0010	0001	0000
	V	Tag	Data		Data	Data	Data	Data	Data	Data	Data	Data
63				…								
62				…								
61				…								
60	1	~~0x0028F~~	???	…	**a**	**b[3]**	**b[2]**	**b[1]**	**b[0]**	**c**	**d**	???
		…	…		…							
1				…								
0				…								

16 words (16x Data) = 64B

Return to example No 1

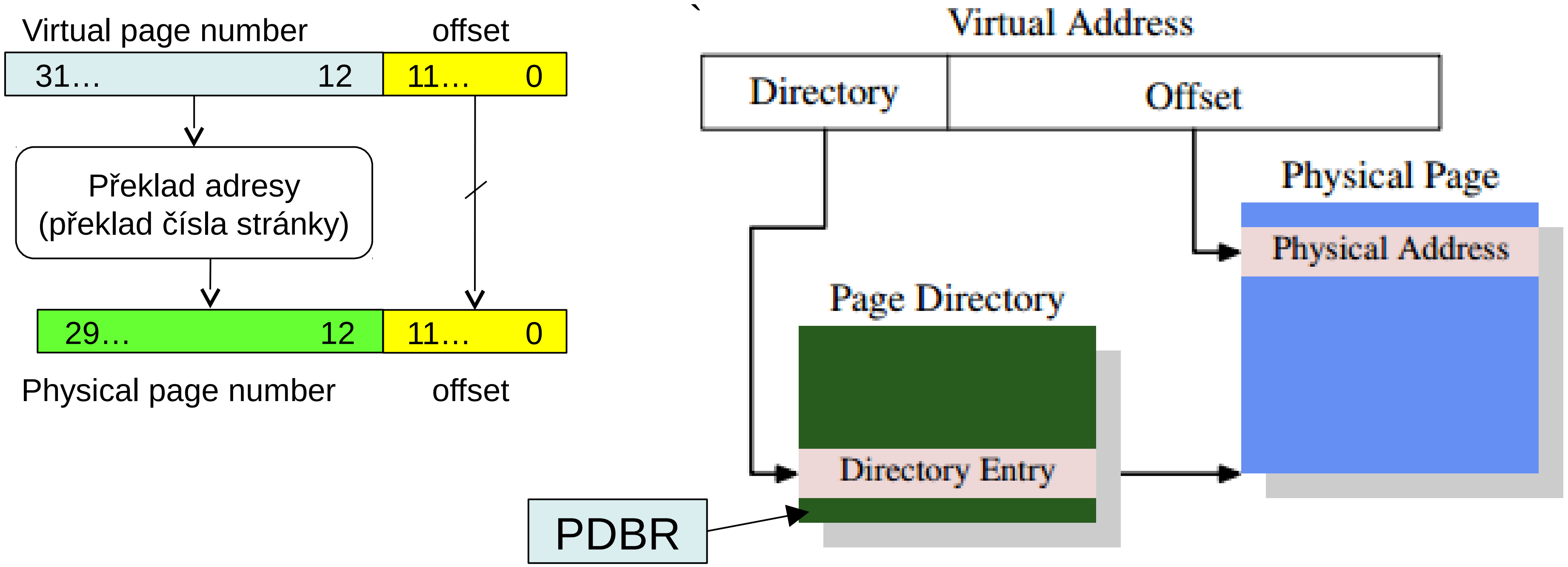
Conclusions:

- Paging (virtual memory realization) does not disturb spatial locality principle => important for cache.
- **Data on adjacent virtual addresses will be stored in physical memory side by side (unless it exceeds the page boundary).**
- If a page fault occurs as a result of the cache miss, then the whole page moves to memory from disc, and then the cache line moves to the cache. The next cache miss inside the page will no longer cause a page fault (until the page is replaced by another page).

Address translation

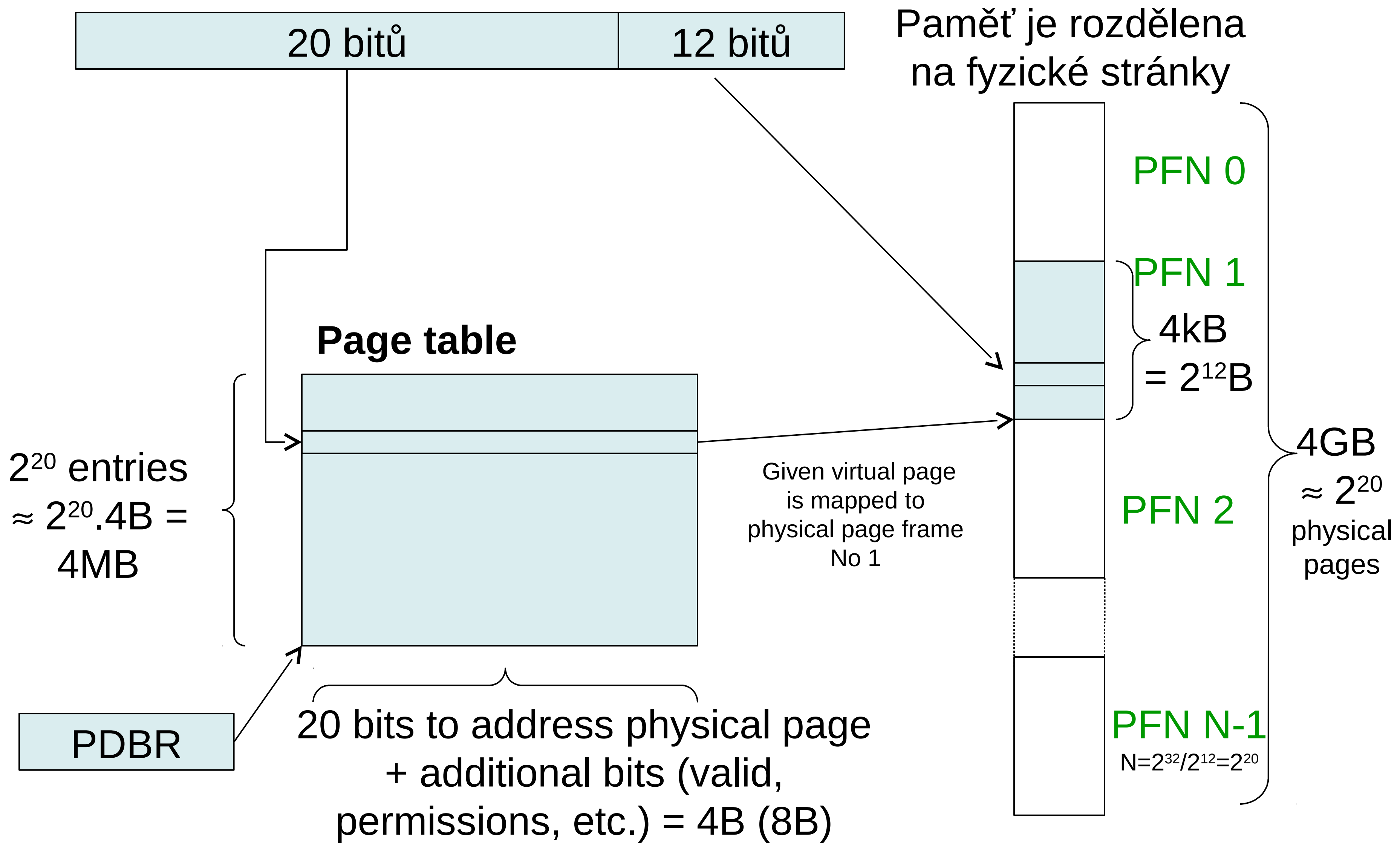
- Page Table
  - Root pointer/page directory base register (x86 CR3=PDBR)
  - Page table directory PTD
  - Page table entries PTE
- Basic mapping unit is a page (page frame)
- Page is basic unit of data transfers between main memory and secondary storage
- Mapping is implemented as look-up table in most cases
- Address translation is realized by **Memory Management Unit** (MMU) which is part of CPU
- Example follows on the next slide:

Virtual to physical address translation realization



- Data structure of Page Directory (Page Table) is stored in main memory. Allocation of continuous area in physical memory and placing its physical address into PDBR register is task of operating system.
- PDBR - page directory base register – x86 is realized by CR3 register – it contain physical address of start of page table
- PTBR - page table base register – the same in another documents…

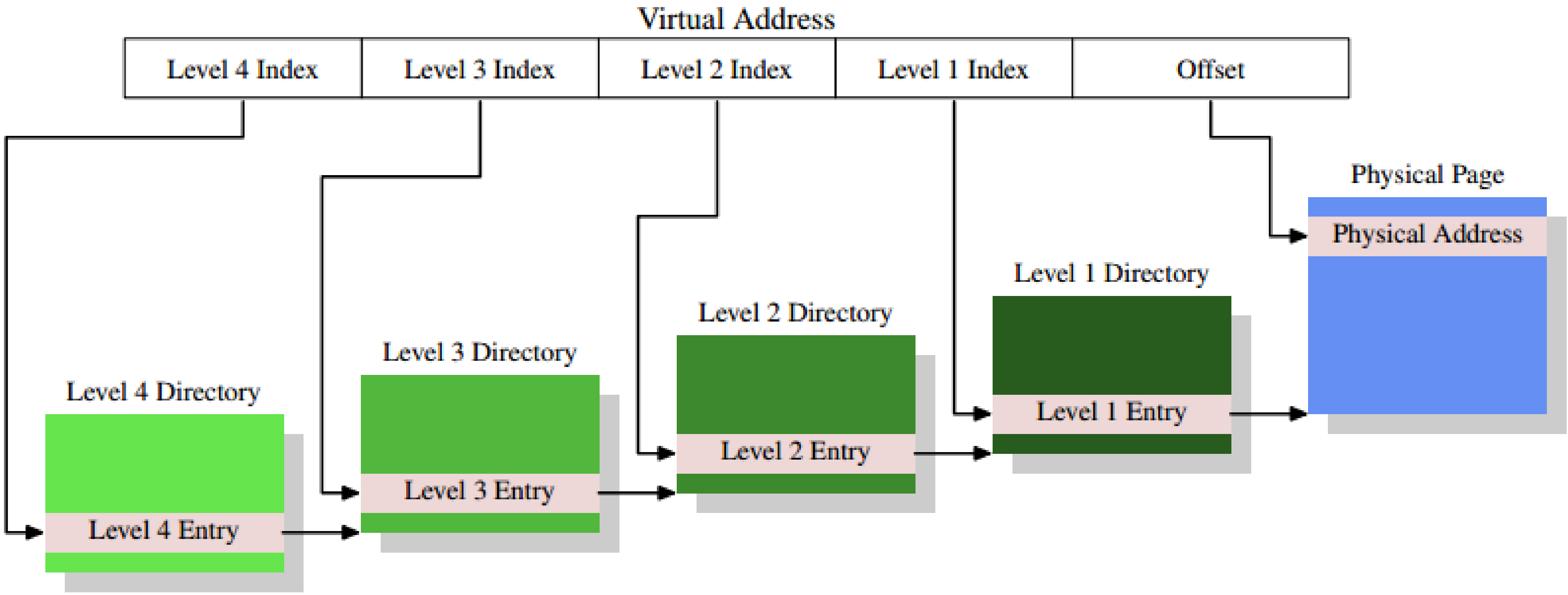
Virtual to physical address translation realization


20 bitů
12 bitů
Paměť je rozdělena na fyzické stránky
PFN 0
PFN 1
4kB = 2^{12}B
PFN 2
PFN N-1
N=$2^{32}/2^{12}=2^{20}$
4GB ≈ 2^{20} physical pages
Page table
2^{20} entries ≈ 2^{20}.4B = 4MB
Given virtual page is mapped to physical page frame No 1
PDBR
20 bits to address physical page + additional bits (valid, permissions, etc.) = 4B (8B)

Analyze memory requirements for page table

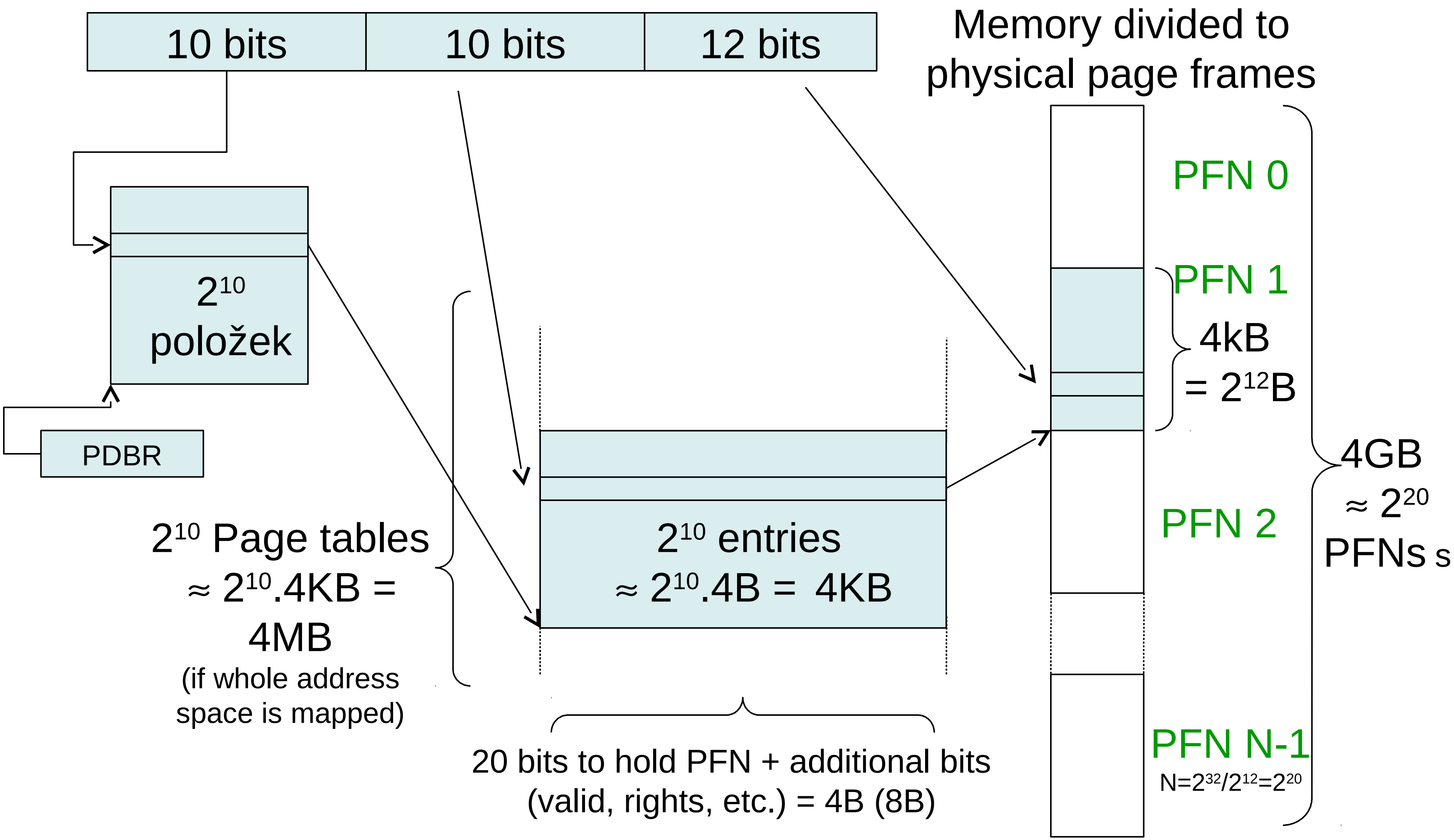
- Typical page size 4 kB = 2^12
- For known page address only 12 bit are used as offset to address inside page. 20 bits (for 32-bit address) remain.
- The fastest map/table look-up is indexing ⇒ use array structure
- Result: Page Directory (Page Table) should provide 2^20 entries (PTEs). This is not practical and causes may disadvantages. For 200 processes it requires 200×2^20×4 bytes = 800 MB of memory.
- Usual process/thread work with small part of the whole address space (temporal locality principle) in given „instant of time“. Usual process utilizes only smaller portion of maximal address space as well.
- Physical space allocation fragmentation problem when large compact table is used for each process
- Solution: multilevel page table – lower levels populated only for used address ranges

Multilevel page table



4-Level Address Translation

Multilevel page table – 2 levels



Multilevel pagetables

Remarks to previous slide:

- Only a few processes uses whole available address space => it is not necessary to allocate all 2^{10} Page tables of the second level
- Page tables can be paged to disk (not used in Linux)

Overall notes:

- Intel IA32 implements 2-level page tables
  - Level 1 Page Table is named as Page Direcory (10 bits for indexing)
  - Level 2 Page Table is named simply Page Table (10 bits)
- For 64-bit virtual addresses is usual to use less bits for physical address – for example 48 or 40 and even virtual address has some limitations.
- Intel Core i7 uses 4-level page tables and 48 address space
  - Level 1 Page Table: Page global directory (9 bits) indexed by bits 39..47
  - Level 2 Page Table: Page upper directory (9 bits) indexed by bits 30..38
  - Level 3 Page Table: Page middle directory (9 bits) indexed by bits 21..29
  - Level 4 Page Table: Page table (9 bits) indexed by bits 12..20

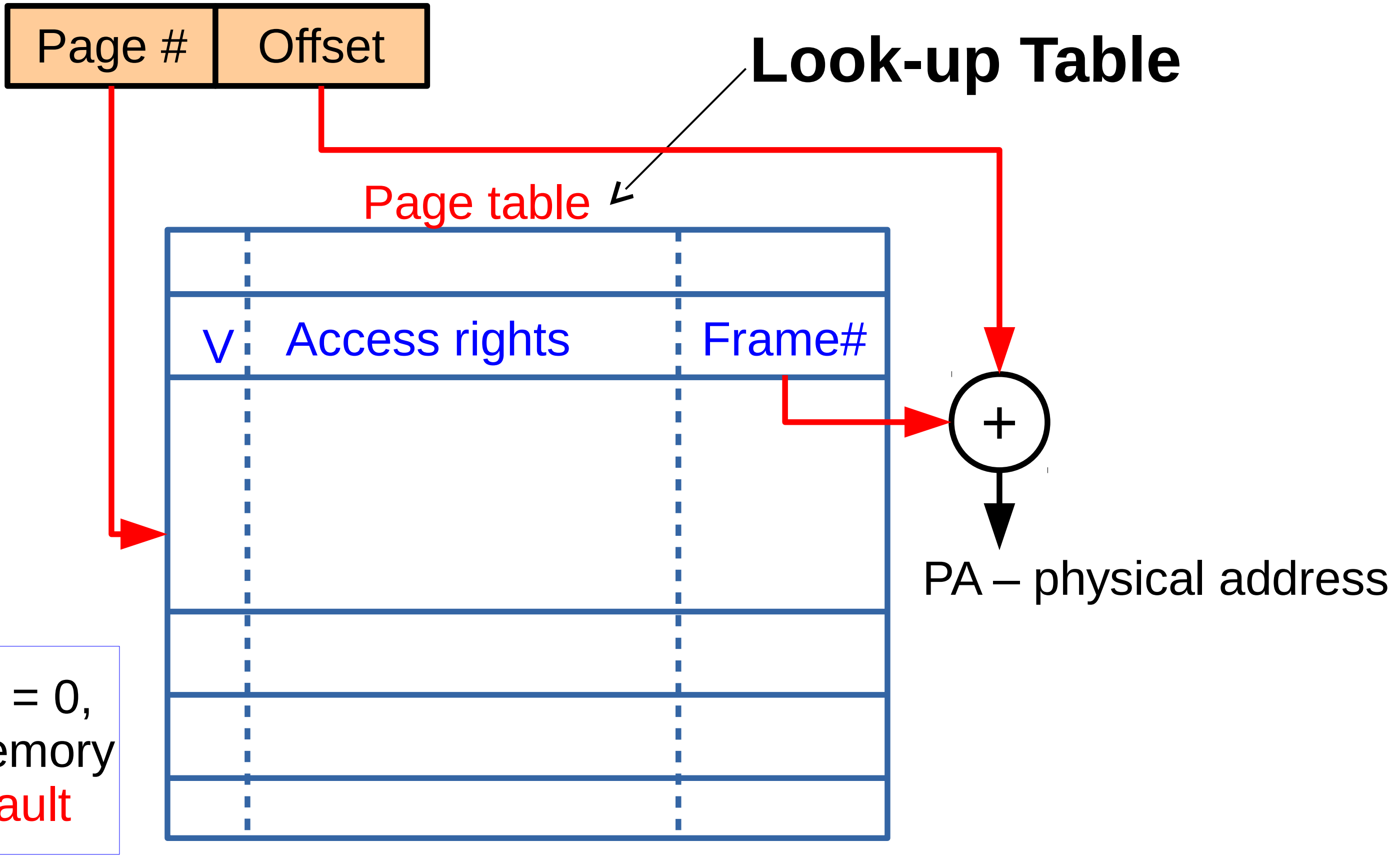
Which fields are in page table entries?

VA – virtual address

Page Table Base Register PTBR

Index into pagetable

Page valid bit – if = 0, page not in the memory results in page fault



Page table placed in physical memory

Which fields are in page table entries?

Analyze entries of Page Directory (Page Table na 1.úrovni)

31…							1	0
Can be used by operating system								P=0

31… 12		6	5	4	3	2	1	0
Base address of Page table		…	A	PCD	PWT	U/S	R/W	P=1

- bit 0: Present bit – informs if page is present in memory(1) or on disc (0) Named as V – valid bit in some other sources/architectures.
- bit 1: Read/Write: if 1 – R/W; if 0 – only read allowed (RO)
- bit 2: User/Supervisor: 1 – user accessible; 0 – only OS
- bit 3: Write-through/Write-back – cache strategy for given page
- bit 4: Cache disabled/enabled – some peripherals are mapped into memory space (memory mapped I/O), this allows immediate read/write of its registers. These addresses can be considered as un-cached I/O ports.
- bit 5: Accessed – set if page content is read/written by CPU – is used during decission which pages should be freed when memory is required.

Which fields are in page table entries?

Analyze entries of (leaf) Page Table (Page Table of level 2 for example)

31…							1	0
Can be used by operating system								P=0

31… 12	7	6	5	4	3	2	1	0
Base address of Page	…	D	A	PCD	PWT	U/S	R/W	P=1

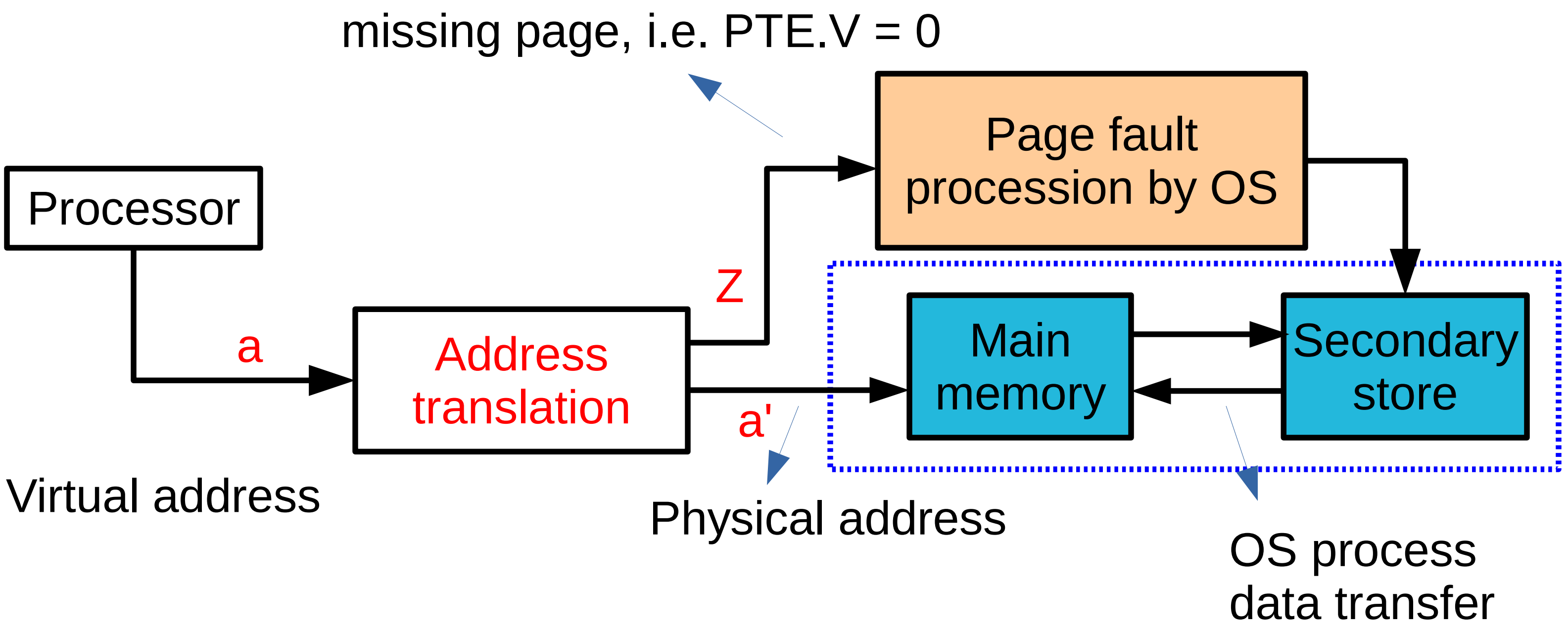
- bit 6: Dirty bit – Is set if page has been modified (written into) after last operating system check. Such page has to be written back to swap in case of PFN reuse for other purposes. Operating system is responsible to clear of Dirty and Accessed bits.

Remarks

- Each process has its own page table
- Process specific value of CPU PTBRT register is loaded by OS when given process is scheduled to run
- This ensures memory separation and protection between processes
- Page table entry format fields required to remember
  - V – Validity Bit. V=0 → page is not valid (is invalid)
  - AR – Access Rights (Read Only, Read/Write, Executable, etc.),
  - Frame# - page frame number (location in physical memory)
  - Other management information, Modified/Dirty, (more bits discussed later, permission, system, user etc.).

V	AR	Frame#

Virtual memory – Hardware and software interaction



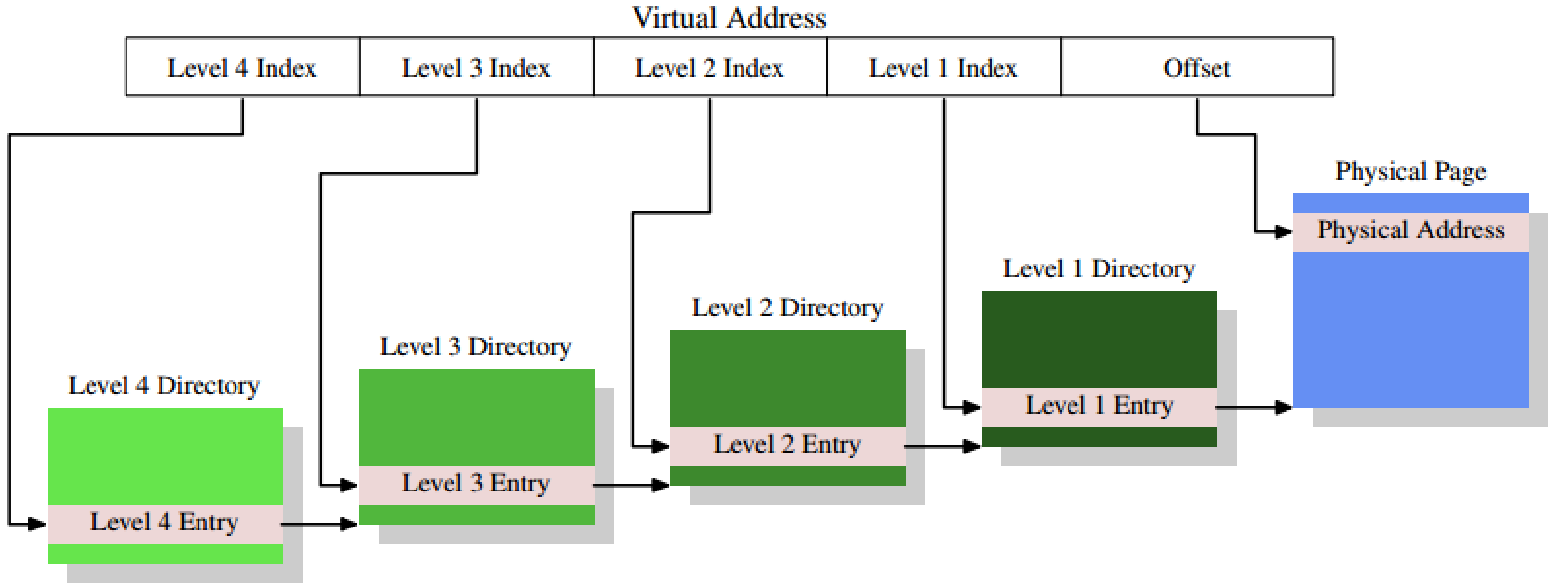
How to resolve page-fault

- Check first that fault address belongs to process mapped areas
- If free physical frame is available
  - The missing data are found in the backing store (usually swap or file on disk)
  - Page content is read (usually through DMA, Direct Memory Access, part of some future lesson) to the allocated free frame. If read blocks, the OS scheduler switches to another process.
  - End of the DMA transfer raises interrupt, OS updates page table of original process.
  - Scheduler switches to (resumes) original process.
- If no free frame is available, some frame has to be released
  - The LRU algorithm finds (unpinned – not locked in physical memory by OS) frame, which can be released.
  - If the Dirty bit is set, frame content is written to the backing store (disc). If store is a swap – store to the PTE or other place block nr.
  - Then continue with gained free physical frame.

Virtual memory and files on disk (secondary memories)…

- Virtual memory extends available “physical” memory by secondary memory space. The pages are automatically swapped/read to/from disc. This can be reused…
- **Mapping of programs and dynamic libraries into memory:**
  - Programs and libraries are stored as binary files (holding instructions and data) in filesystem
  - When new program is about to be run (process is allocated):
    - OS notes at which virtual address ranges/areas (VMA) should be blocks of file available in given address space
    - OS actualizes process Page table as result of fault and uses information noted in VMAs to fill pages by right content from file then sets entries as Valid=1. In case of physical memory pressure discards unmodified pages ans sets Valid=0
  - Program is read automatically by memory management as it runs…
  - See **mmap()** – functions allocates VMA and update Page table such way that area is transparent window into part of whole file.

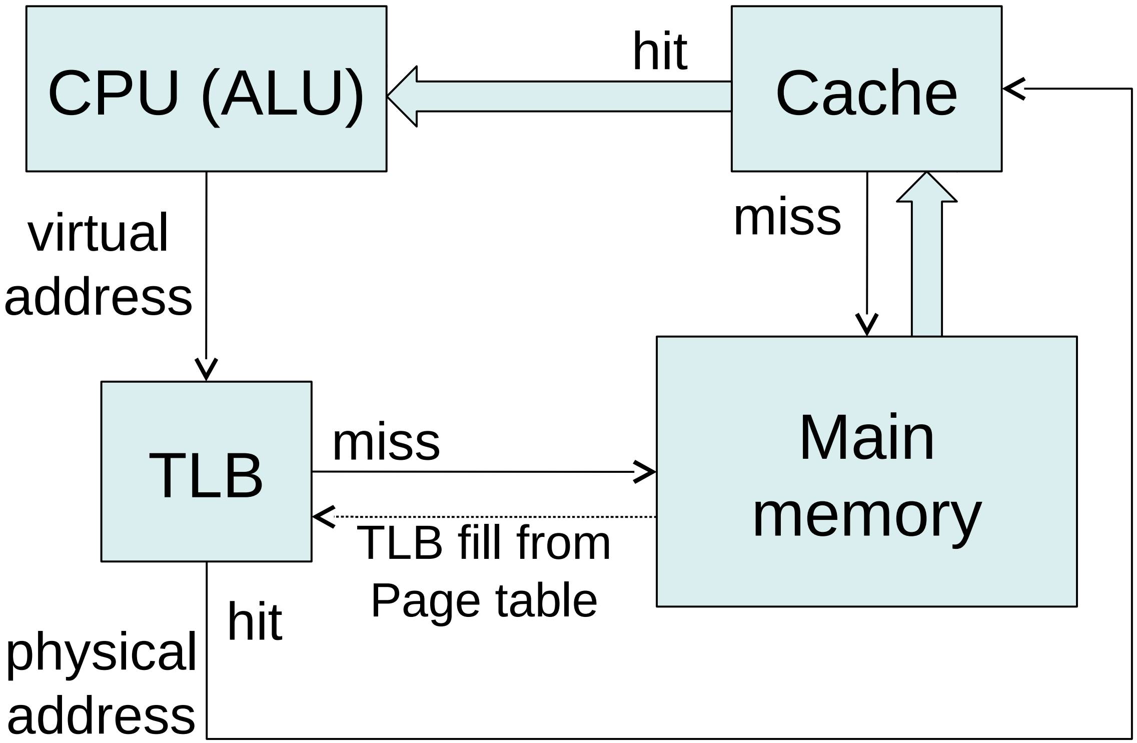
Multilevel page table – translation overhead



4-Level Address Translation

- Translation would take long time, even if entries for all levels were present in cache. (One access per level, they cannot be done in parallel.)
- The solution is to cache found/computed physical addresses
- Such cache is labeled as Translation Look-Aside Buffer
- Even multi-level translation caching are in use today

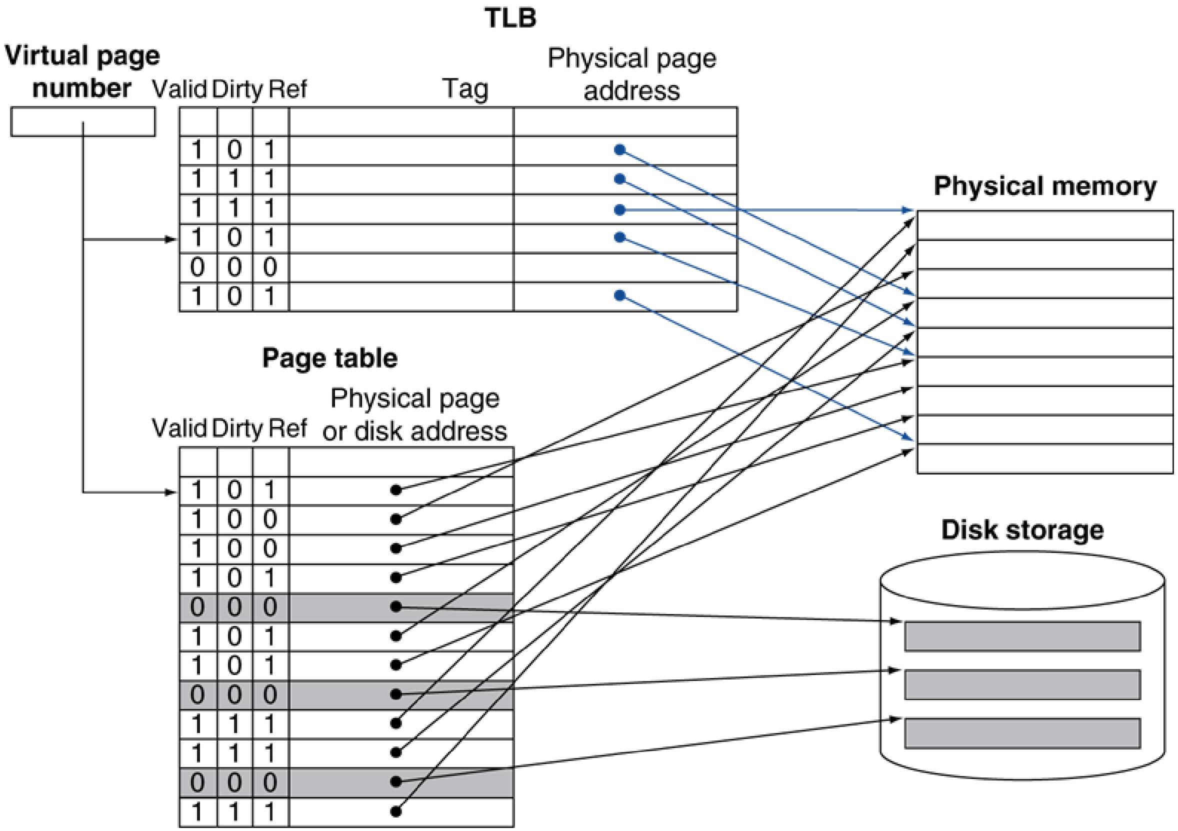
Ideal translation case when TLB serves all translations



- Notice that single memory access can result in multiple misses
- If TLB miss occurs, it is necessary to run HW (or SW on some architectures) *page walk*. It usually uses cached access to page table.

Fast MMU/address translation using TLB

- Translation-Lookaside Buffer, or may it be, more descriptive name – Translation-Cache
- Cache of frame numbers where key is page virtual addresses (virtual page frame number – VPFN)



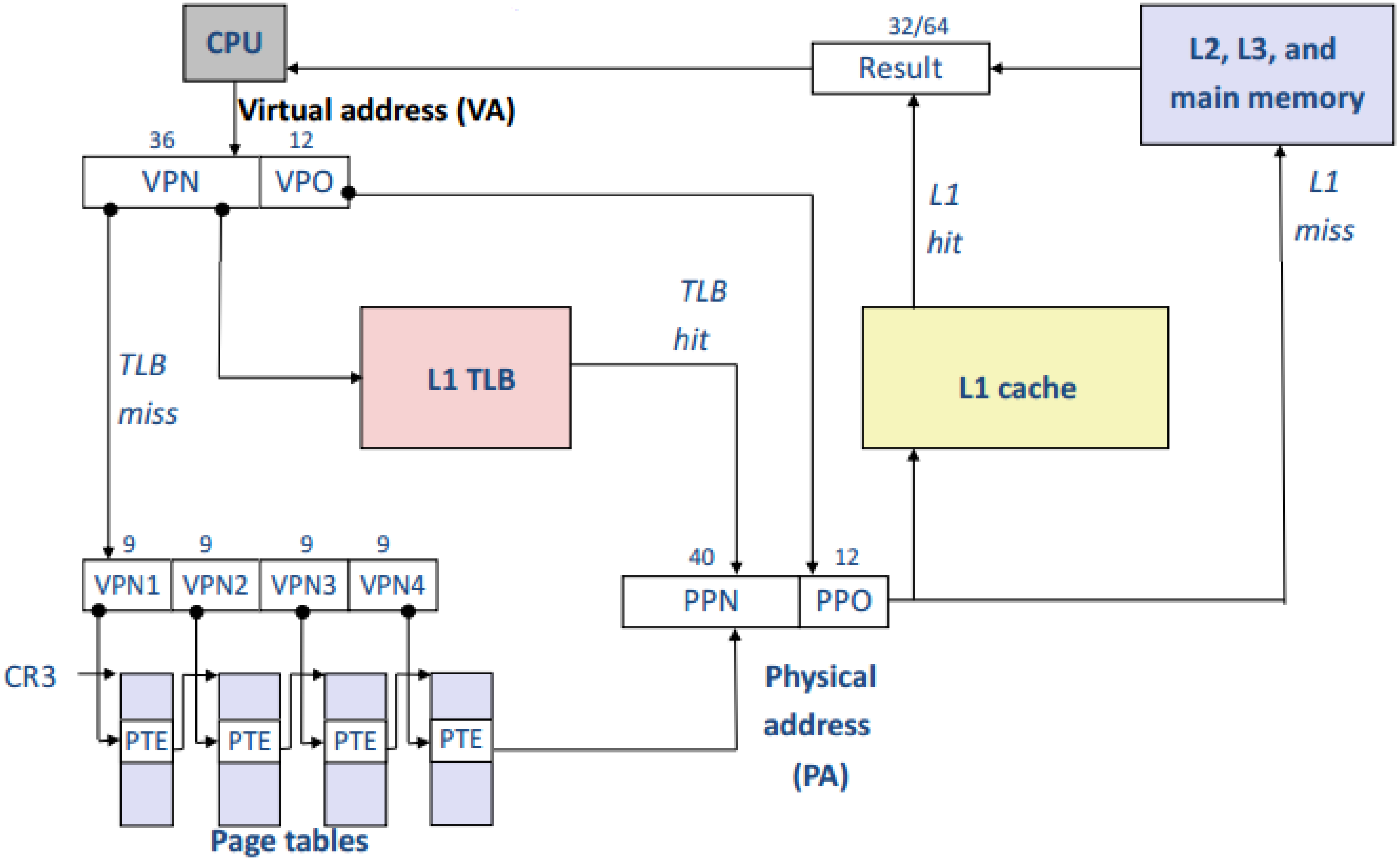
TLB

Valid	Dirty	Ref	Tag	Physical page address
1	0	1		●
1	1	1		●
1	1	1		●
1	0	1		●
0	0	0		
1	0	1		●

Page table

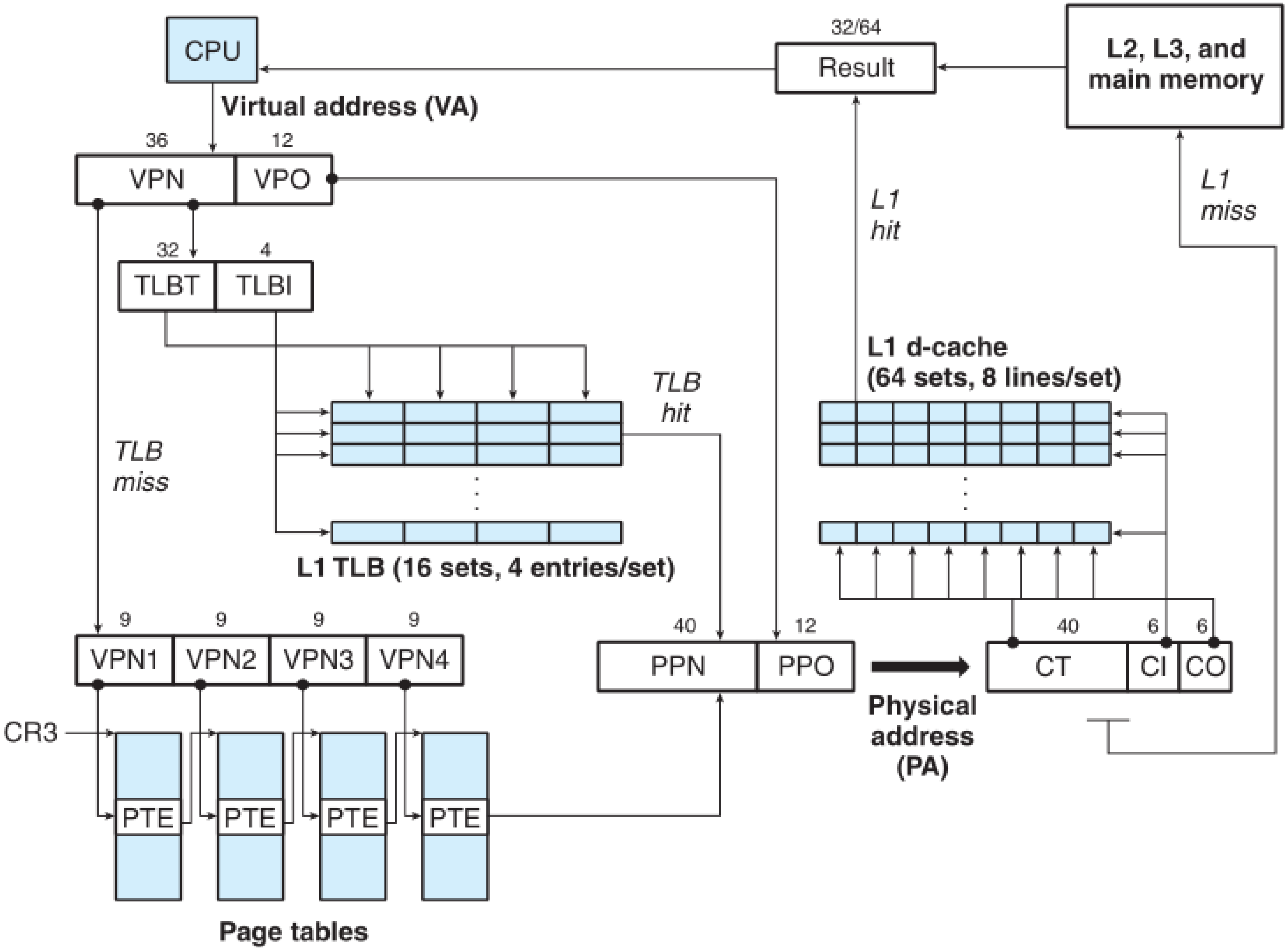
Valid	Dirty	Ref	Physical page or disk address
1	0	1	●
1	0	0	●
1	0	0	●
1	0	1	●
0	0	0	●
1	0	1	●
1	0	1	●
0	0	0	●
1	1	1	●
1	1	1	●
0	0	0	●
1	1	1	●

Address stranslation – Intel Nehalem (Core i7)

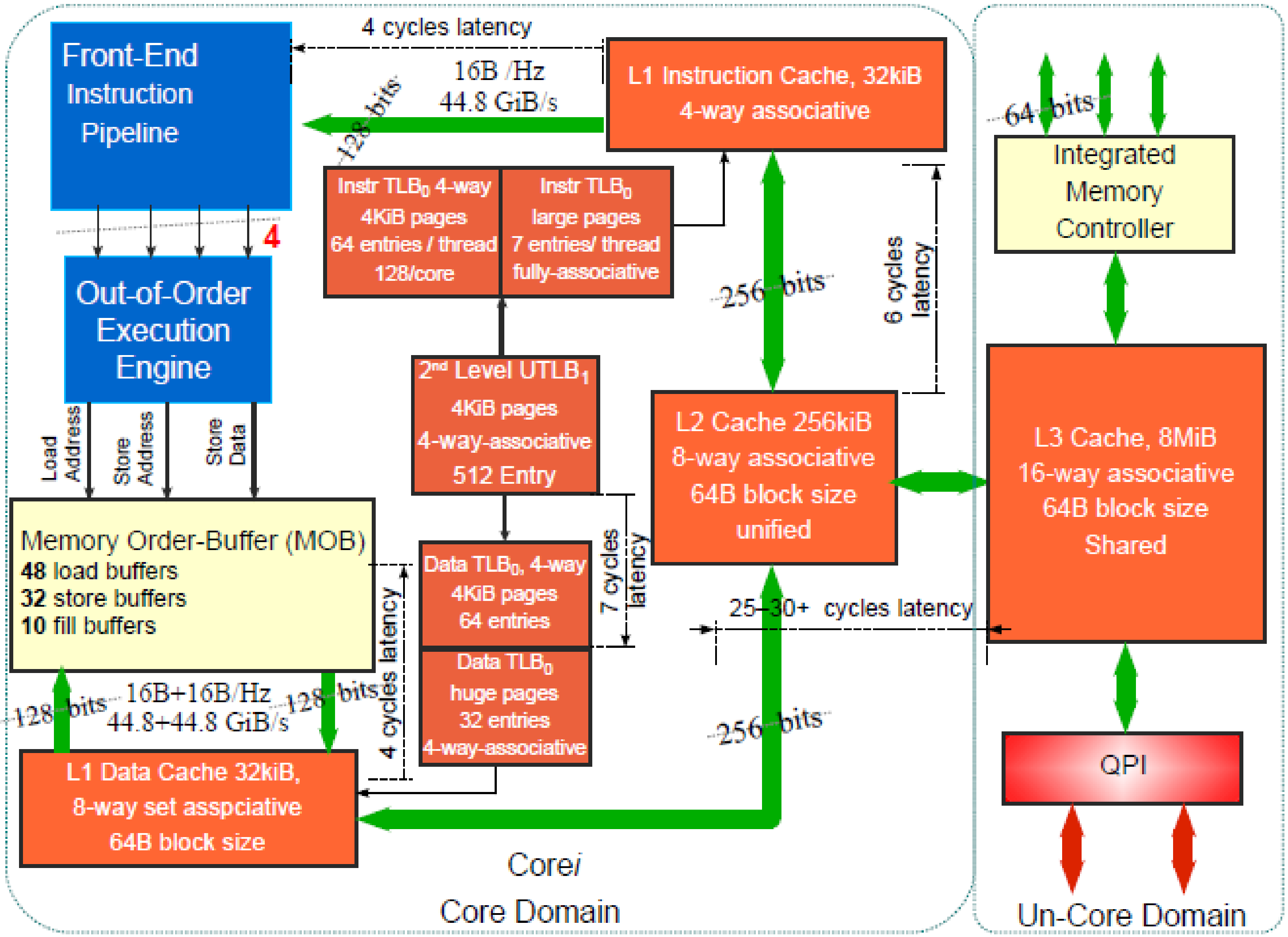


http://cs.nyu.edu/courses/spring13/CSCI-UA.0201-003/lecture18.pdf

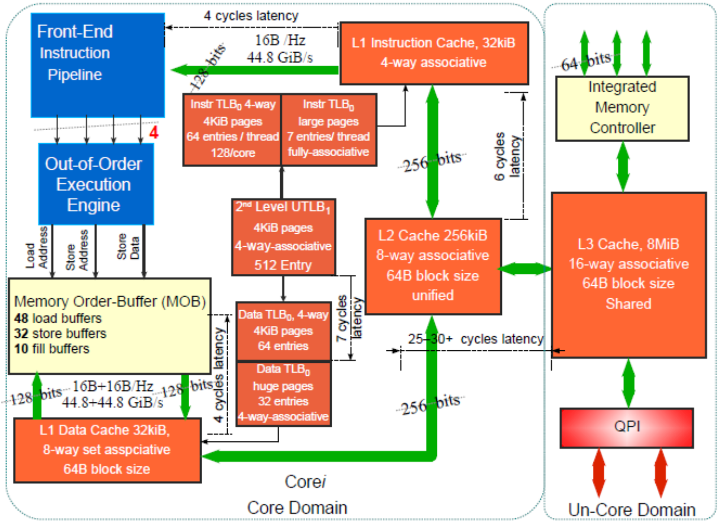
Address translation – Intel Nehalem (Core i7) – in more detail



Memory organization - Intel Nehalem (Core i7)



Memory organization - Intel Nehalem – some remarks

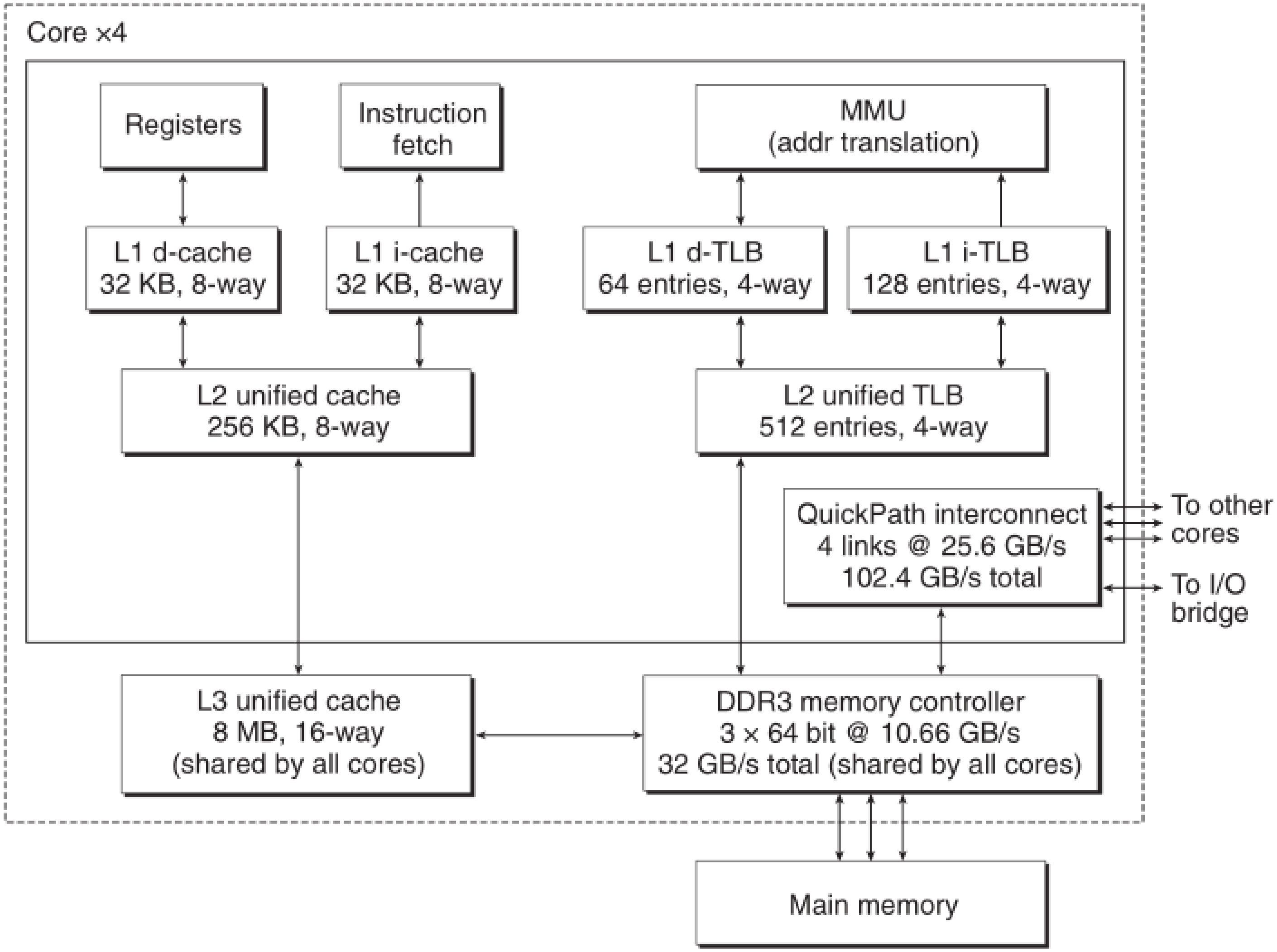

Front-End Instruction Pipeline
4 cycles latency
16B /Hz
44.8 GiB/s
128 bits
L1 Instruction Cache, 32kiB 4-way associative
4
Instr TLB$_0$ 4-way 4KiB pages 64 entries / thread 128/core
Instr TLB$_0$ large pages 7 entries/ thread fully-associative
256 bits
6 cycles latency
Out-of-Order Execution Engine
2nd Level UTLB$_1$ 4KiB pages 4-way-associative 512 Entry
Load Address
Store Address
Store Data
L2 Cache 256kiB 8-way associative 64B block size unified
Memory Order-Buffer (MOB)
48 load buffers
32 store buffers
10 fill buffers
Data TLB$_0$, 4-way 4KiB pages 64 entries
7 cycles latency
25–30+ cycles latency
Data TLB$_0$ huge pages 32 entries 4-way-associative
128 bits
16B+16B/Hz
44.8+44.8 GiB/s
128 bits
4 cycles latency
256 bits
L1 Data Cache 32kiB, 8-way set asspciative 64B block size
Core*i*
Core Domain
64 bits
Integrated Memory Controller
L3 Cache, 8MiB 16-way associative 64B block size Shared
QPI
Un-Core Domain

- Block size: 64B
- CPU reads whole cache line/block from
  main memory and each is 64B aligned
- (6 LS bits are zeros), partial line fills allowed
- L1 – Harvard. Shared by two (H)threads
  instruction – 4-way 32kB, data 8-way 32kB
- L2 – unified, 8-way, non-inclusive, WB
- L3 – unified, 16-way, inclusive (each line stored in L1 or L2 has copy in L3), WB
- Store Buffers – temporal data store for each write to eliminate wait for write to the cache or main memory. Ensure that final stores are in original order and solve “transaction” rollback or forced store for:

  - exceptions, interrupts, serialization/barrier instructions, lock prefix,..
- TLBs (Translation Lookaside Buffers) are separated for the first level

  Data L1 32kB/8-ways results in 4kB range (same as page) which allows to use 12 LSBs of virtual address to select L1 set in parallel with MMU/TLB

Intel Core i7 – the same but different view



Typical sizes of today I/D and TLB caches comparison

	Typical paged memory parameters	Typical TLB
Size in blocks	16 000-250 000	40-1024
Size	500-1 000 MB	0,25-16 KB
Block sizes in B	4 000-64 000	4-32
Miss penalty (clock cycles)	10 000 000 – 100 000 000	10-1 000
Miss rates	0,00001-0,0001%	0,01-2
Backing store	Pages on the disk	Page table in the main memory
Fast access location	Main memory frames	TLB

More efficient memory use – means to speed up programs

Your program can take into account the page size and use memory more efficiently - by aligning to the multiples of page size, and then reducing internal and external page fragmentation .. (ordering allocations etc. See also *memory poo*l)

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
   printf(„Page size id: %ld B.\n",
          sysconf(_SC_PAGESIZE));
   return 0;
}
```

Allocation of memory aligned to some block size:

```
void * memalign(size_t size, int boundary)
void * valloc(size_t size)
```

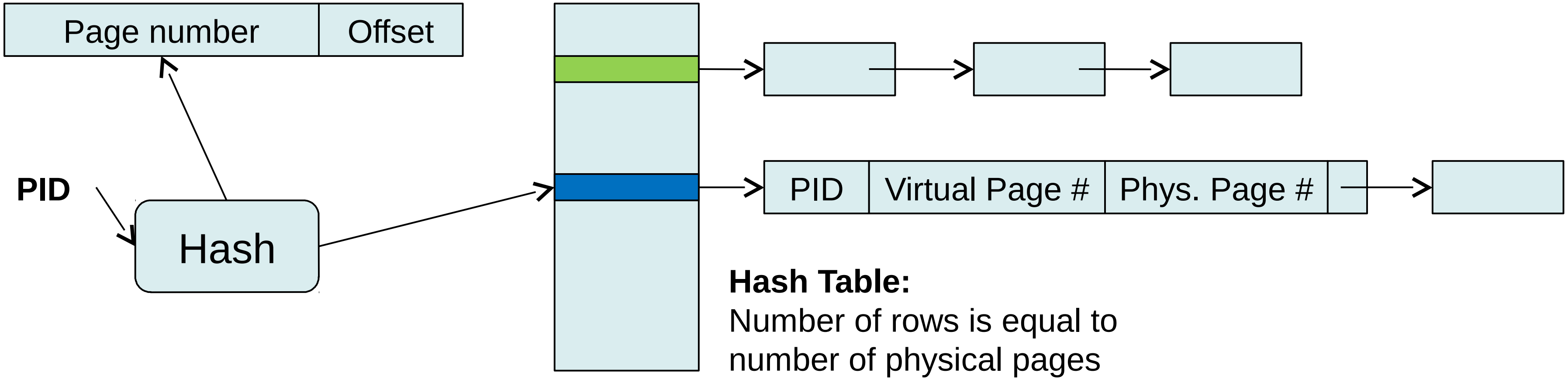
windows

```
#include <stdio.h>
#include <windows.h>

int main(void) {
    SYSTEM_INFO s;
    GetSystemInfo(&s);
    printf("Page size is: %ld B.\n",
       ns.dwPageSize);
    printf("Address range for application (and dll):
       0x%lx – 0x%lx\n",
       s.lpMinimumApplicationAddress,
       s.lpMaximumApplicationAddress);
    return 0;
}
```

Are **hierarchical (multilevel) page tables** only alternative?

- Hierarchical page tables are (in fact) represent a tree structure that needs to be searched
- Another alternative exists: **Inverted Page Tables**
- 64-bit virtual address space is quite large, physical memory is much smaller -> big disproportion
- **Idea:** Physical memory is divided into pages. It is enough to have array of rows equivalent to numebr of physical pages to store information to which virtual page is physical one allocated.
- The problem is poor spatial locality (cacheability) as has result and limitation to physical map page only to single virtual one

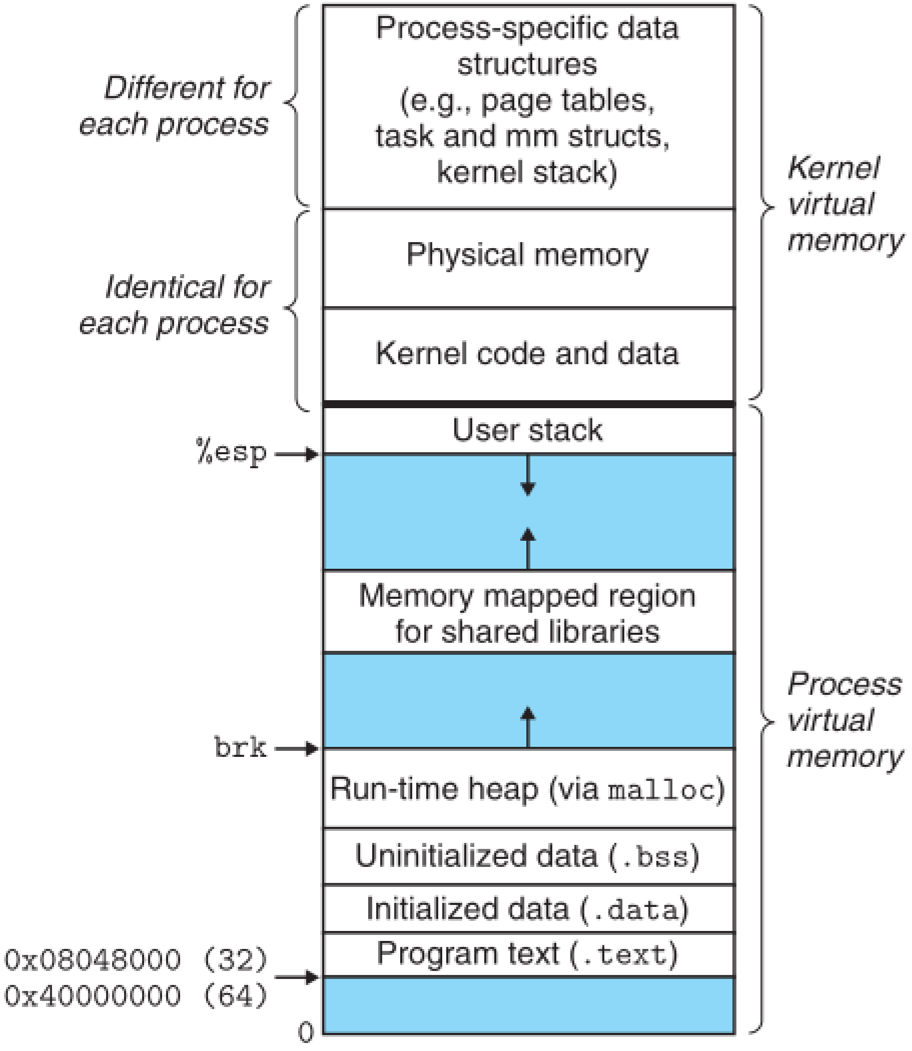


Hash Table:
Number of rows is equal to
number of physical pages

Virtual memory as used in Linux

Definitions – put things into context

- Linux organizes VM as collection of **virtual memory areas (VMA)**
- **Area** is continuos block valid process virtual memory, which has some purpose. Example: code segmant, data segment, heap, shared library segment, user stack.
- **Each valid virtual page belong to some VMA.**
- Use of areas/segments allows to organize virtual memory with „gaps“ – segments are not required to follow up each other. Even higher levels of page tables can be populated at runtime.



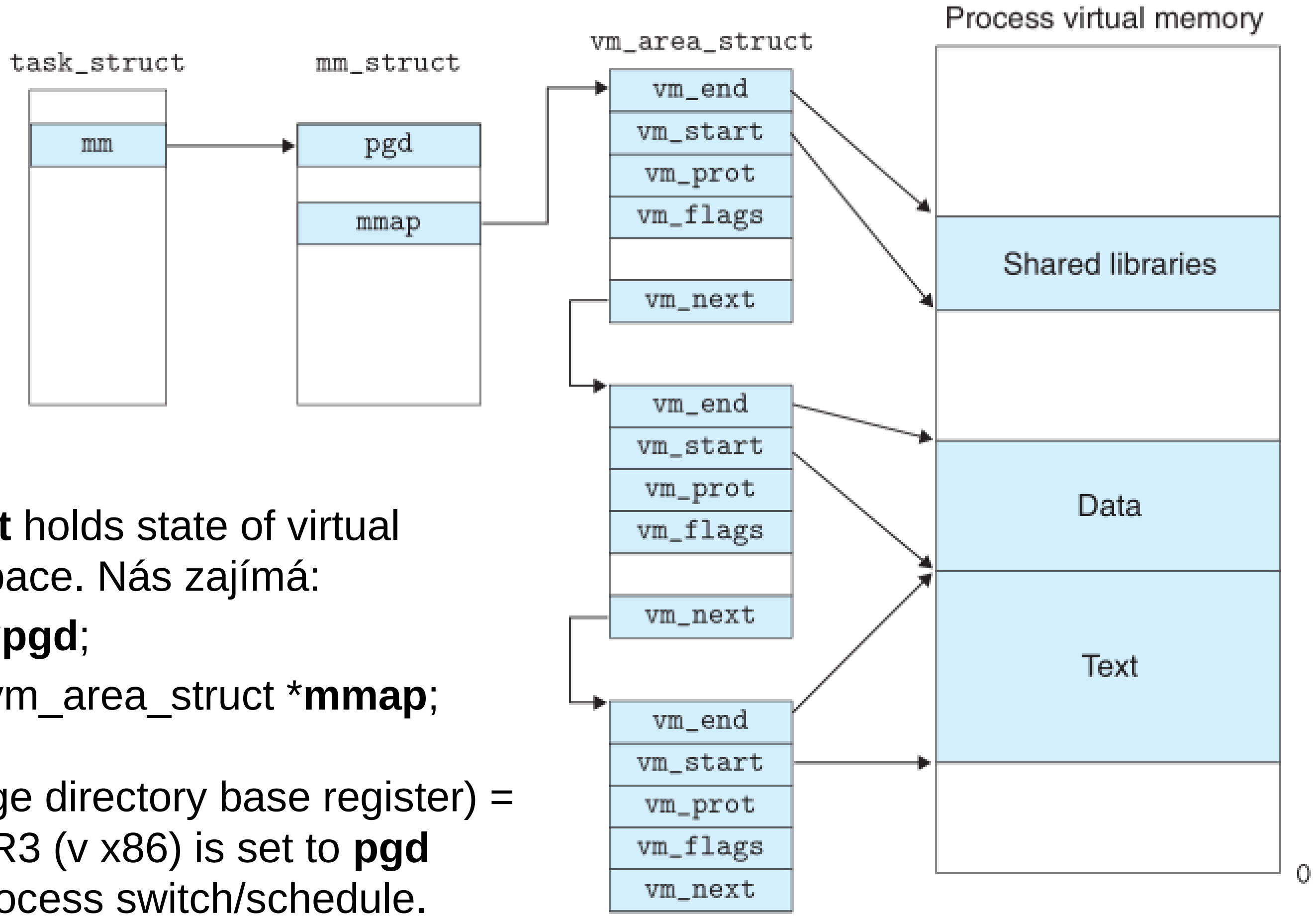
Different for each process
Identical for each process
Process-specific data structures (e.g., page tables, task and mm structs, kernel stack)
Physical memory
Kernel code and data
Kernel virtual memory
User stack
%esp
Memory mapped region for shared libraries
brk
Run-time heap (via malloc)
Uninitialized data (.bss)
Initialized data (.data)
Program text (.text)
0x08048000 (32)
0x40000000 (64)
0
Process virtual memory

struct task_struct

```
struct task_struct {
  volatile long        state; /* -1 unrunnable, 0 runnable, >0 stopped */
  long                counter;
  long                priority;
  unsigned            long signal;
  unsigned            long blocked;    /* bitmap of masked signals */
  unsigned            long flags;      /* per process flags, defined below
        */
  int errno;
  long                debugreg[8];    /* Hardware debugging registers */
  struct exec_domain   *exec_domain;
  struct linux_binfmt  *binfmt;
  struct task_struct   *next_task, *prev_task;
  struct task_struct   *next_run,  *prev_run;
  unsigned long        saved_kernel_stack;
  unsigned long        kernel_stack_page;
  int                exit_code, exit_signal;
  unsigned long        personality;
  int                dumpable:1;
  int                did_exec:1;
  int                pid;
  int                pgrp;
  int                tty_old_pgrp;
  int                session;
  int                leader;
  int                groups[NGROUPS];
struct task_struct   *p_opptr, *p_pptr, *p_cptr,
                   *p_ysptr, *p_osptr;
struct wait_queue    *wait_chldexit;
unsigned short       gid,egid,sgid,fsgid;
  unsigned long        timeout, policy, rt_priority;
  unsigned long        it_real_value, it_prof_value, it_virt_value;
  unsigned long        it_real_incr, it_prof_incr, it_virt_incr;
  struct timer_list    real_timer;
  long               utime, stime, cutime, cstime, start_time;
unsigned long        min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
  int swappable:1;
  unsigned long        swap_address;
  unsigned long        old_maj_flt;    /* old value of maj_flt */
  unsigned long        dec_flt;        /* page fault count of the last time */
  unsigned long        swap_cnt; *number of pages to swap on next pass */
struct rlimit        rlim[RLIM_NLIMITS];
  unsigned short       used_math;
  char               comm[16];
int                link_count;
  struct tty_struct    *tty;           /* NULL if no tty */
struct sem_undo      *semundo;
  struct sem_queue     *semsleeping;
struct desc_struct *ldt;
struct thread_struct tss;
struct fs_struct     *fs;
struct files_struct  *files;
struct mm_struct *mm; /* memory management info */
struct signal_struct *sig; /* signal handlers */
};
```

struct task_struct

- **task_struct** contains (or better points) information which allows kernel to manage execution of process (PID, pointer to user stack user stack,…). **mm_struct *mm** is important for us now.



mm_struct holds state of virtual memory space. Nás zajímá:

- pgd_t ***pgd**;
- struct vm_area_struct ***mmap**;

PDBR (page directory base register) = PTBR = CR3 (v x86) is set to **pgd** value at process switch/schedule.

Page Fault Exception Handling - simplified

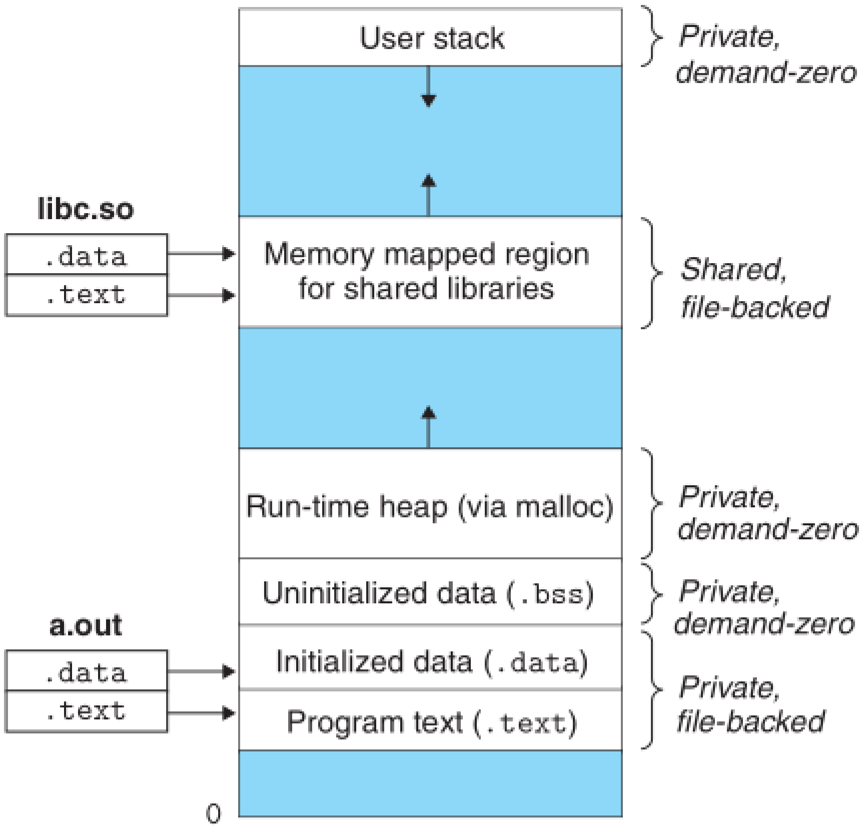
Consider that MMU (Mem Manag Unit) invokes **Page Fault** as result of attempt to access some memory location/translate its virtual address A (not present in TLB). This results in execution of Page Fault Handler:

- It check if A is valid. i.e. it A points within some VMA defined by vm_area_struct. vm_start and vm_end limits are checked. Sequential search of VMAs list is time consumpting => there is kept up to date search RB tree for each process areas.
  If address is not valid for process -> **Segmentation Fault** and kill
- If attempt is valid is operation permitted? access rights (read, write, execute). If not -> **Protection Exception** and kill the process
- Access is legal to legal address. Free or victim physical page has to be found and released (marked as invalid in appropriate page table(s) and if dirty write it back to disk), load new/requested page content, actualize Page Table. Finis and return from Page Fault Handler. CPU restarts instruction causing Page Fault. MMU serves address A translation correctly this time – without raising Page Fault.

Memory Mapping

Linux initialize content of (area) of virtual memory by:

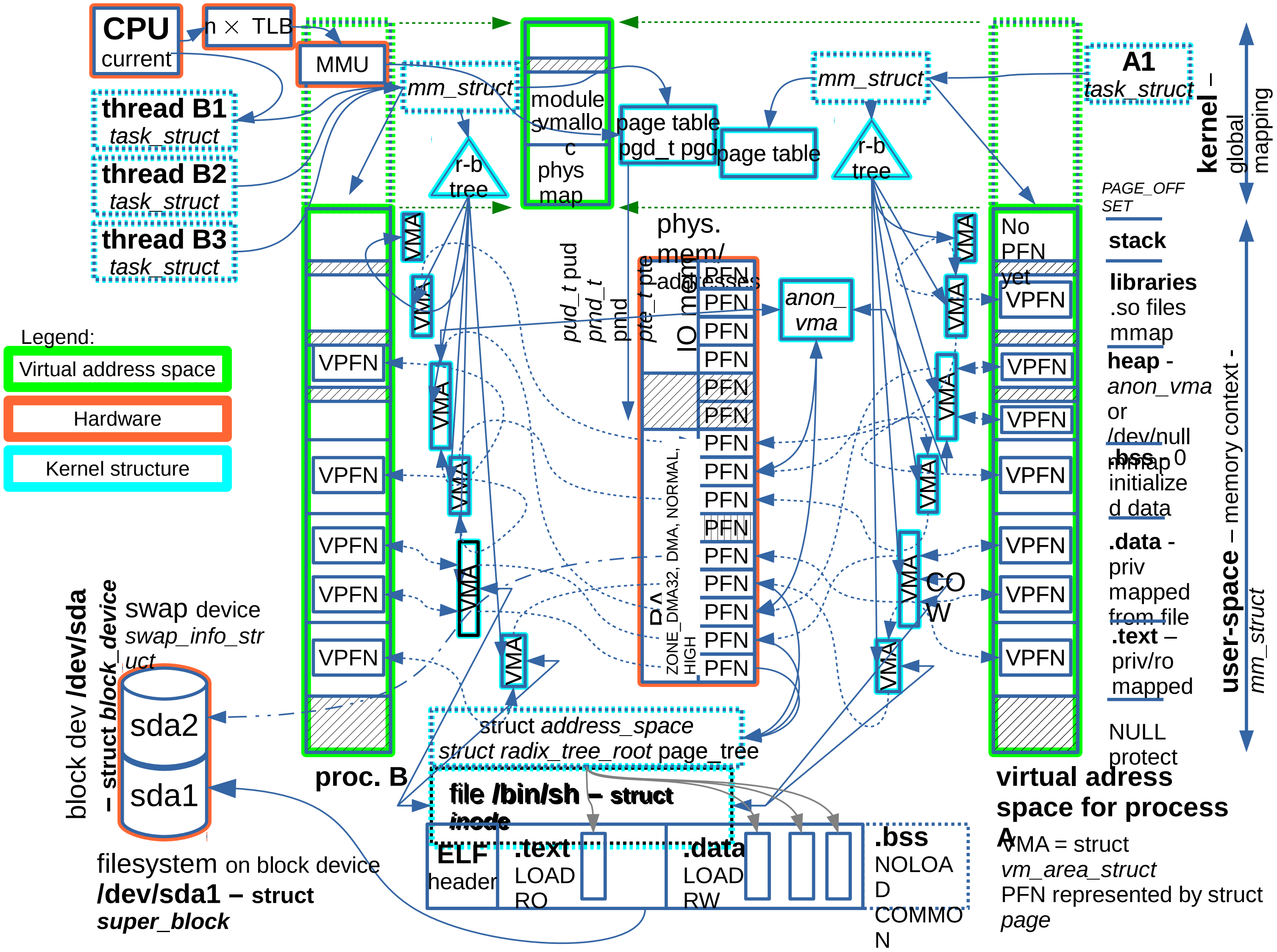
- **Regular file** (read from area result in read from the file)
- **Anonymous file/area** – if CPU read from given address the first time, kernel uses RO mapping to the global zero initiated page. For write it searches for free page or releases some (if it is dirty it is written to *swap file*), copies zero page, actualize Page Table. These initially zeroed pages are sometimes labeled as *demand-zero pages*



More programmer use in *mmap()* function

- It maps files or devices into process address space to be accessed directly by CPU

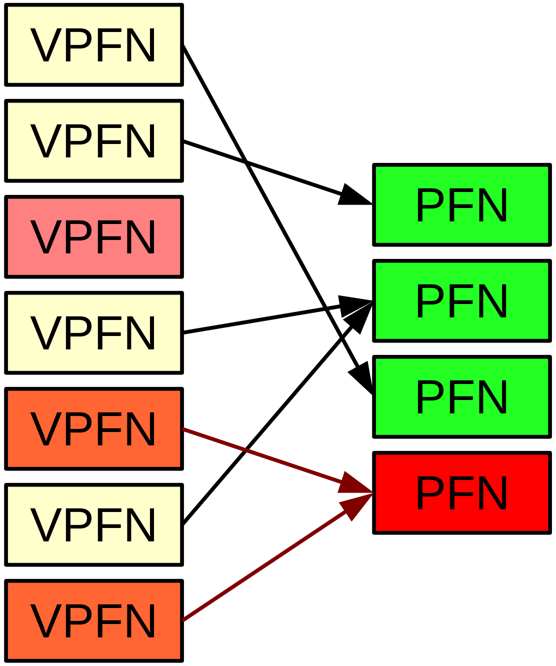
Linux memory management structures



See the description in the notes of the OpenDocument format

Significant problem with reverse pages mapping

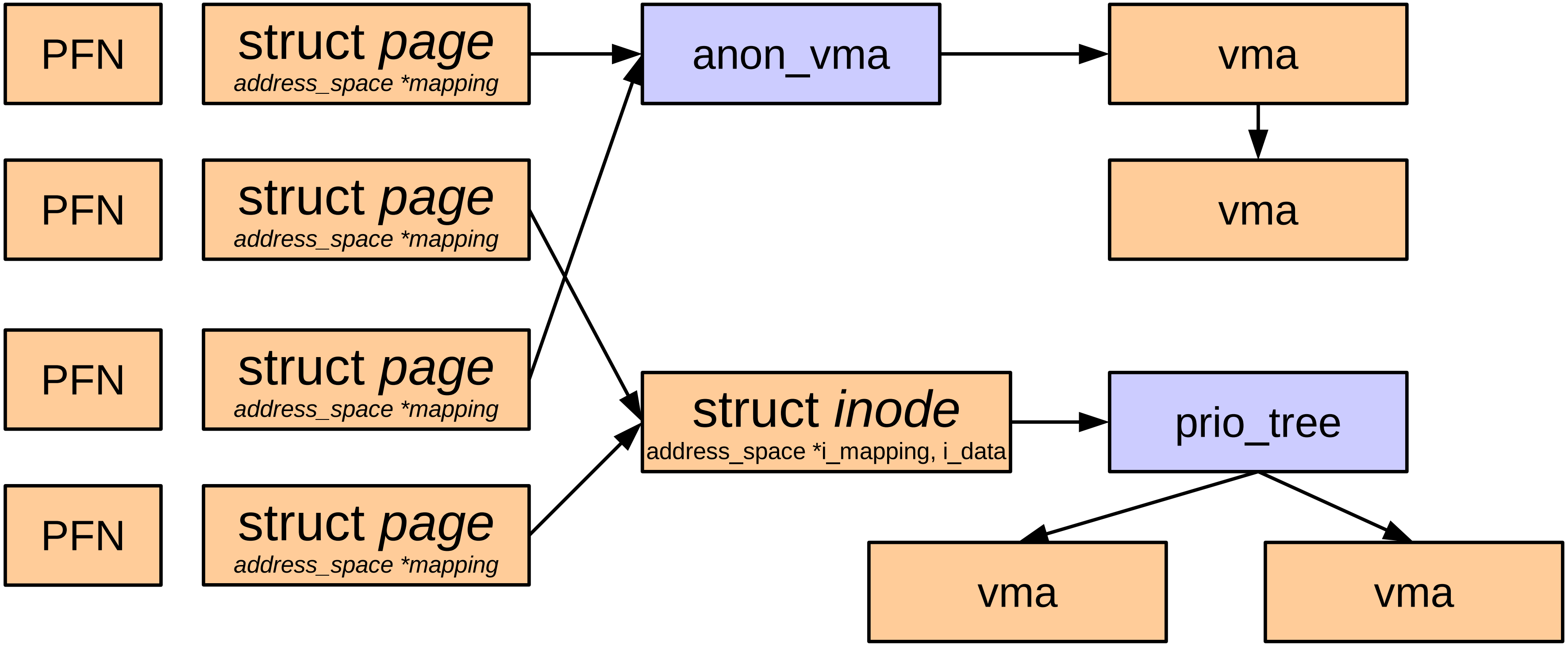
- Multiple virtual pages (VPFN) from single or even multiple processes can map to single physical page (PFN).
- The mapping of virtual to physical page costs only one entry (PTE) in the page table (4/8 byte) + some much smaller amount in upper table levels (PGD, PUD, PMD) + one are description (*vm_area_struct*) for whole range.
- Physical pages are critical resource and each is described by its *struct page* which is found directly from its location page-frame-number (PFN)
- PFN = virtual_address >> PAGE_SHIFT



- Location of all PTE for given PFN is complicated but required to manage and release/free physical page (for reuse) and invalidation all corresponding PTEs.
- If record is held for each page then it requires 8 bytes per list entry (next and PTE pointer) but there is only ~900M lowmemory on x86 in 32-bit
- If 1000 processes maps 2G shared memory (code, same data) then lists for reverse mapping take 1000*2*1024*1024*1024/4096*8/1024/1024/1024 = 3.9 GiB

Structures used for reverse mapping

- Solution: objrmap + anon_vma + prio_tree
- Each physical page (page struct) points only to corresponding VMA or inode



- PTE is then found by searching given page in VMA , finding its offset. VMA points to *mm_struct* which defines page table for memory context. Location of PTE in page table is easy from VMA start and page offset in VMA.

Virtual memory are data structures

```
struct vm_area_struct {
      struct mm_struct * vm_mm;     /* The address space we belong to. */
      unsigned long vm_start;       /* Our start address within vm_mm. */
      unsigned long vm_end;         /* The first byte after our end address
                                 within vm_mm. */

      /* linked list of VM areas per task, sorted by address */
      struct vm_area_struct *vm_next;

      pgprot_t vm_page_prot;        /* Access permissions of this VMA. */
      unsigned long vm_flags;       /* Flags, see mm.h. */

      struct rb_node vm_rb;

      union {
            struct {
                  struct list_head list;
                  void *parent;     /* aligns with prio_tree_node parent */
                  struct vm_area_struct *head;
            } vm_set;

            struct raw_prio_tree_node prio_tree_node;
      } shared;

      struct list_head anon_vma_node;     /* Serialized by anon_vma->lock */
      struct anon_vma *anon_vma;    /* Serialized by page_table_lock */

      /* Function pointers to deal with this struct. */
      const struct vm_operations_struct *vm_ops;

      /* Information about our backing store: */
      unsigned long vm_pgoff;       /* Offset (within vm_file) in PAGE_SIZE
                                       units, *not* PAGE_CACHE_SIZE */
      struct file * vm_file;        /* File we map to (can be NULL). */
      void * vm_private_data;       /* was vm_pte (shared mem) */
      unsigned long vm_truncate_count;/* truncate_count or restart_addr */
};
```

A file's MAP_PRIVATE vma can be in both i_mmap tree and anon_vma list, after a COW of one of the file pages. A MAP_SHARED vma can only be in the i_mmap tree. An anonymous MAP_PRIVATE, stack or brk vma (with NULL file) can only be in an anon_vma list.

For areas with an address space and backing store, linkage into the address_space->i_mmap prio tree, or linkage to the list of like vmas hanging off its node, or linkage of vma in the address_space-> i_mmap_nonlinear list.

Some system calls and their interaction with memory

- **fork**() – creates new process as a copy of calling one. All pages (except of SHM) of parent are marked as Copy-On-Write (COW) and are shared between processes – VMA are kept with original writable state but PTE of parent and child are marked RO => only the first write result in page separation/copying for child or parent and marking it RW
- **clone**() – create new process, but allows to control if memory management and other aspect should be separated or shared with parent process or thread => if MM shared then thread is created
- **mmap**() – creates new VMA/region in linear/virtual address space of given process and allows to map file into it
- **mremap**() – remaps or modifies size and attributes of memory region
- **munmap**() – releases whole or part of region. (If unmapped in middle, region is divided into two)
- **shmat**() – connects/maps shared memory segment to the process
- **shmdt**() – undoes shmat()
- **exit**() – destroys process and all its memory areas and regions

References

- Randal E. Bryant, David R. O’Hallaron: Computer Systems, A Programmer’s Perspective.
- http://cs.nyu.edu/courses/spring13/CSCI-UA.0201-003/lecture18.pdf
- David Money Harris and Sarah L. Harris: Digital Design and Computer Architecture, Second Edition. Morgan Kaufmann.