Memory Cells and Registers

- Variables can be stored in “memory cells”, i.e., they reside in the main memory, i.e., access is slow and typically uses caches.

- Variables can also reside in internal memory of processors called “registers”, i.e., access is then immediate. But the number of registers is very limited, e.g., 16 registers in your x86-64/AMD64 computer.

- Remark: C has the (ancient) keyword register which is recommendation to the compiler to store the variable in a register for fast access, e.g., as loop index. Today obsolete since compilers will optimize for themselves and may even ignore register. W.r.t. language they will still honor that the adress & of a register variable cannot be taken.
Memory Consistency and Cache Protocols
Introduction: What do we expect of a parallel programme?

(x, y memory cells, r1, r2 registers, always initialized with 0)

// P1
x=1
r1=y

// P2
y=1
r2=x

// r1==?, r2==?

// naively (r1==1; r2==1)
// - real parallel computers often give other results

// Depending on the consistency model the results
// (r1==1; r2==1), (r1==0; r2==1) and (r1==1; r2==0);
// may be allowed. Even (r1==0; r2==0) may be allowed.

Remark: The case (r1==0; r2==0) is allowed in Intel 64/AMD64!
Access to the main DRAM memory (Load/Store) is orders of magnitudes slower than arithmetic-logical operations performed by processors:

⇒ Use small, expensive, fast buffers (“caches”).

Example: Intel Xeon Nehalem memory latency: Main memory (DRAM = dynamic random access memory) 80 cycles, L1 cache (SRAM = static RAM) 3 cycles.

Due to the locality of memory accesses it is sufficient to use caches that are magnitudes smaller than the main memory.

Example: Core-i7 Nehalem L1 cache size: 64KB.

Today’s processors indeed use a hierarchy of caches (L1, L2, L3)

Example: Core-i7 Nehalem: L1=4 cycles, L2=10 cycles, L3=49 cycles, DRAM=74 cycles.

Cache organization (cache line, cache associativity, etc.), see Computer Architecture I+II.
Caches in Intel Broadwell (2014-...)

- 14nm shrink of 22nm Haswell, i.e., basically same architecture as Haswell
- L1 cache: 64 KB (=32KB data+32KB instruction) per core; data: 8-way associative, write-back
- L2 cache: 256 KB per core, 8-way associative, writeback
- L3 cache: may vary (2 MB to 20 MB), shared among cores

Details can be found in “Intel 64 and IA-32 Architectures Optimization Reference Manual”, Sept. 2014


On Linux systems

```
grep . /sys/devices/system/cpu/cpu0/cache/index*/*
```

will give you information on the caches of your machine.
### Cache Parameters of the Haswell Microarchitecture (2013)

<table>
<thead>
<tr>
<th>Cache Level</th>
<th>1st (data)</th>
<th>1st (instr)</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity</td>
<td>32KB</td>
<td>32KB</td>
<td>256KB</td>
<td>Can vary</td>
</tr>
<tr>
<td>Associativity</td>
<td>8-way</td>
<td>8-way</td>
<td>8-way</td>
<td>Can vary</td>
</tr>
<tr>
<td>Line Size</td>
<td>64 bytes</td>
<td>64 bytes</td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td>Fastest Latency</td>
<td>4 cycles</td>
<td>11 cycles</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Throughput</td>
<td>0.5 bytes/cycle</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peak Bandwidth</td>
<td>64 (load) + 32 (store)</td>
<td>64 bytes/cycle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Update Policy</td>
<td>Writeback</td>
<td></td>
<td>Writeback</td>
<td>Writeback</td>
</tr>
</tbody>
</table>
**Associative Memory (German: Assoziativspeicher): Content Addressable Memory (CAM)**

In standard random access memory (RAM) the memory is presented an address, and it returns the data stored in this address.

- CAM does not use addresses but uses part of the content (denoted “tag”) to find stored data.
- The tag is searched in a table of stored data, and then the full matching data is returned.
- It is possible that no entry matches.
- If several entries match they are all returned.
Cache Line

The portion of data that is exchanged between RAM and cache (or between levels of Caches).

- A cache line is valid or invalid completely (see MESI).
- There are no partially valid cache lines.
- Even if a CPU wants to read only a single byte the complete cache line of 64 bytes is loaded into the cache.
- Fortunately, RAM is usually faster in providing cache lines that follow each other in memory (prefetch/burst).

Ways (German: Weg)

The number of ways is the number of places where a cache line can be stored.

- In a 8-way set associative cache, a given cache line can be stored in 8 different places in the cache.
- A set associative cache with only one way is also called a directly mapped cache.
- In a directly mapped cache a given cache line can be stored in only a single place of the cache. Will result in many collisions since other cache lines have to be stored in the same place.
Set (German: Satz)

In a set associative cache a cache line can be stored in a set of locations.

- A set has a number (set address).
- The number of locations that a cache line can be stored in is called “ways”.
- An memory address is divided into a tag the set address and the byte address.
Cache Organization (here: 32-bit Pentium from 1993)

The 66 MHz Intel Pentium of 1993 was a 32 bit processor, produced in an 800nm process and famous for the fdiv bug. [Remark: Reincarnated, extended to 64bit, as Intel Xeon Phi with 60 cores. Rank 1 of the current TOP500 uses 3M Xeon Phis.]

**2-way set associative L1 cache with 128:** 8 KB = 2 \cdot 32 \cdot 128 bytes (ways*line*sets):

<table>
<thead>
<tr>
<th>31</th>
<th>20</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tag (20 bits)</td>
<td>Set Address (7 bits) = values: 0...127</td>
<td>Byte Address (5 bits) = values: 0...31</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache-Directory</th>
<th>Cache-Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>Way 0</td>
</tr>
<tr>
<td>0</td>
<td>Tag</td>
</tr>
<tr>
<td>1</td>
<td>Tag</td>
</tr>
<tr>
<td>2</td>
<td>Tag</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>126</td>
<td>Tag</td>
</tr>
<tr>
<td>127</td>
<td>Tag</td>
</tr>
</tbody>
</table>

The set address is the number of the set in the cache directory. The byte adress is the number of the entry in the cache line.

Haswell 2014: 8-way associative with 32Kb size; cache line size of 64 bytes.
Discussion of Set Associative Cache

- Compromise between a fully associative cache, and a directly mapped cache.
- Fully associative caches require parallel searches for the cache line in all slots (very hard to build).
- Directly mapped cache uses can store a given cache line in only one slot but may cause many collisions.

Consider a parking lot with 1000 spots for the 6000 students of the TUBAF.

- Directly-mapped cache: You have only one possible parking lot based in the last 3 digits of the student id ⇒ You only have to check one spot but it is likely that someone has taken it (whose car you then have to push out :)).
- Fully associative cache: You can park anywhere ⇒ You will eventually find an empty spot but you have to search the complete parking lot.
- 10-way set associative cache: You can park in the 10 places that are identical to the last 2 digits of your student id. ⇒ Good: You only have to check 10 places. Bad: All 10 spots may be taken (you push out the one that has parked the longest; Least Recently Used (LRU) strategy)
Cache Replacement Strategy

There are different replacement strategies for caches lines

- Directly mapped caches do not need a cache replacement strategy (there is only a single possible location).
- LRU (Least Recently Used) strategy such that “old” entries are replaced first: access to a set in way 0 = LRU\(_0\) = 1 and LRU\(_0\) = 0; access to a set in way 1 = LRU\(_1\) = 0 and LRU\(_1\) = 1.
- Random replacement (often not worse than LRU-strategy): a way is randomly chosen.
Cache Performance

Caches are quite effective today

- Miss rates of 5-10% for L1 and
- less than 1% for L2 are commonly cited.
- But code as

```c
double a[m][n];
double norm;
for(int j=0;j<m;++j)
    for(int i=0;i<n;++i)
        norm+=a[i][j]^2
```

can give miss rates of 100%. 
Cache Performance in Matrix-Matrix-Multiplication

Loops for

\[ c_{ij} = \sum_k a_{ik} b_{kj} \]

can be ordered \( ijk, kij, jki \). Test the performance!! But beware:

```bash
oliver@hilbert:~/cpplab> time ./a.out
real   0m11.866s
user   0m10.534s
sys    0m0.017s
oliver@hilbert:~/cpplab> g++ -O matmult.c
oliver@hilbert:~/cpplab> time ./a.out
real   0m0.370s
user   0m0.367s
sys    0m0.001s
oliver@hilbert:~/cpplab> g++ -O3 matmult.c
oliver@hilbert:~/cpplab> time ./a.out
real   0m0.002s
user   0m0.000s
sys    0m0.001s
```
Usual definition of cache coherence

- Processor P1 **writes** the value \( w \) to a memory cell and afterwards **reads** from this position. If P1 then **reads** from the same memory cell it will **read** the value \( w \) (and not the old value) – if not another processor has **written** to the same the memory cell.
- P1 **writes** the value \( w \) to the memory; P2 **reads** from the same memory cell afterwards; then P2 **reads** the value \( w \) if the time is \( > 0 \) (alternatively: if the time is \( > \varepsilon \)).
- All processors observe the **same order** of writes to a memory position. (This implies that writes to different memory positions may be reordered.)

Cache coherence for multi processors

- Coherence with snooping ("Schnüffeln"): Memory addresses of writes are observed (on the memory bus).
  - "Write through"-protocols
  - "Write back"-protocols: Example **MESI = Write back Protokoll**
  - ...
- Central directories
  - ...


Simple “Write through”-Protocol

Notation:  
hit = is in the cache  
miss = is not in the cache

read miss: Request is served from the main memory
read hit: Request is served from the cache
write miss: Update of the main memory (“Write through”)
write hit: Update in the cache and the main memory, remote cache entries are invalidated

Can lead to bad performance since write operations always “write through” to the main memory.

⇒ Better performance with “Write back” protocols.
Intel MESI (a “Write back”-protocol)

4 states “Modified (M), Exclusive (E), Shared (S), Invalid (I)”

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid (I)</td>
<td>Cache-entry invalid</td>
</tr>
<tr>
<td>Exclusive (E)</td>
<td>Cache-entry valid, main memory up-to-date, no copies exist</td>
</tr>
<tr>
<td>Shared (S)</td>
<td>Cache-entry valid, main memory up-to-date, copies may exist</td>
</tr>
<tr>
<td>Modified (M)</td>
<td>Cache-entry valid, main memory invalid, no copies exist</td>
</tr>
</tbody>
</table>

Transition between states: **Cache line** is the smallest entity that is managed by a cache, e.g., 128 bytes.

**read hit:** Processor reads a cache line which exists in its cache

**write hit:** Processor writes a cache line which exists in its cache

**read miss:** Processor reads a cache line which does not exist in its cache

**write miss:** Processor writes a cache line which does not exist in its cache

**remote read miss:** Another processor reads a cache line which does not exist in its cache but that exists in the cache of a different processor.
**Example:**

- P1 reads a cache line, the cache line is loaded to the cache ⇒ state **E** (“Exclusive”). Further read accesses by P1 ⇒ P1 remains in state **E** (“Exclusive”).
- P2 requests read from a memory cell of the cache line. Cache line is loaded to the cache ⇒ state **S** (“Shared”) for P1 and P2.
- Further read requests by P1 or P2 ⇒ remain in state **S** (“Shared”).
- P2 writes to the cache line ⇒ state **I** (“Invalidate”) for P1 and state **M** (“Modified”) for P2 - but write to the main memory is postponed (write-back cache!).
- If P1 requests a read from the memory cell then it must wait until P2 has written the cache line to the main memory. Afterwards both processors are in state **S** (“Shared”).
Memory consistency models

In case of a single thread the natural order of the program defines an order of operations.

In the case of several threads this is no longer the case. Consistency has to be declared. Weaker consistency allows more aggressive optimization but are more difficult to program.

Theoretical and implemented memory consistency models are subject of ongoing research in academia and industry.

The most important (classic) memory consistency models are

- Sequential consistency (SC) [Lamport 1979]
- Processor consistency (PC) [Goodman 1989]
- Weak consistency (WC)
- ...

Lamport (1979): How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. 2p.

Remark: Intel x86/64 is not SC, but “Total Lock Order + Causal Consistency (TLO+CC)”, where SC ⊂ TSO ⊂ TLO+CC. According to [4] PC ⊃ TSO, since TSO allows W→R and Read-Own-Write-Early. PC allows additionally Read-Other’s-Write-Early.
Sequential Consistency (SC)

**Definition (SC):** The result of a program is the same as if all operations of all processors are executed in a (any!) sequential order. Additionally, the operations of any single processor appear in the order of the program in this sequence.

Two Aspects: Operations executed in some sequential order (program order aspect) Memory operations of each process in program order ("write atomicity").

This means that all processors agree on the order of all memory accesses.

Example 1 (allowed under SC):

x, y memory cells, r1, r2, r3 register, always initialized with 0

```plaintext
// Processor 1          // Processor 2          // Processor 3
x=1 // M1              y=2 // M2              r1=y // M3
r2=x // M4              r3=x // M5

// naive result: r1==0, r2==1, r3==1

//
// SC allows:
// r1==2, r2==0, r3==1, according operation sequence
// M2 M3 M4 M1 M5
```
Sequential Consistency (SC)

Example 2 (not allowed under SC):

```plaintext
// P1 // P2 // P3 // P4
x=100 // M1 x=200 // M2 r1=x // M3 r3=x // M5
r2=x // M4 r4=x // M6

r1==100, r2==200, r3==200, r4==100; // NOT allowed under SC! P3 and P4 see
--- --- // different orders of the
--- --- // write accesses to memory

// An allowed result is
// r1==200, r2==200, r3==200, r4==200; order of operations: M1 M2 M3 M4 M5 M6
```

Remark to the programmer: Languages and compilers will not solve the problem for the programmer! OpenMP does not even guarantee SC: “[...] each thread is allowed to have its own temporary view of the memory.” (OpenMP Specification 2.5/3.1)

Need explicit command to obtain a consistent view of the memory (flush).
Remark: Optimizations in Uniprocessors

Compilers or processors do change the order of operations ("Out-of-order execution")!

The order

Write A
Read A
Write B
Read B

gives the same result as

Write A
Write B
Read A
Read B

The compiler or processor may use an order which results in faster execution.
Connection to optimizations

Relaxed memory consistency leaves more room for optimization. In case of SC the machine has to behave only as if the operations were executed in a (any!) sequential order.

Example: “Read-own-write-early” (is allowed under SC, “early” means ”earlier than others”).

Chronology in the computer:

```
// P1      // P2
 t=1:     x=1
 t=2:     r1=x   r2=x
```

Naively you would expect: \((r1==1, r2==1)\). SC allows that the write \(x=1\) becomes visible to processor P2 only after a delay (e.g., because of write buffering).

The result \((r1==1, r2==0)\) looks as if

```
// P1      // P2
 t=1:     r2=x
 t=2:     x=1
 t=3:     r1=x
```

had been executed. This is allowed in SC.
**SC is too strong!**

- SC disallows many compiler optimizations, i.e., reordering of *any* memory operations!
- SC disallows many hardware optimizations, i.e., store buffers, nonblocking reads
Cache Coherence and Sequential Coherence (SC)

Cache coherence as defined above and implemented, e.g., in MESI, does NOT automatically lead to Sequential Consistency (SC).

In SC the same order of write operations to all memory positions is requested (not only in case of a common write position).

Cache coherence means that writes will eventually be visible to all processors and writes to the same location are observed in order.

Memory models define the bounds on when the value is propagated to other processors: in SC all reads and writes have to be ordered in program order.

Example:

```
// P1
x=1
r1=y

// P2
y=1
r2=x
```

// Cache coherence allows r1==0,r2==0,
// since x,y are different memory cells
// This is not allowed in SC!
Processor Consistency (PC) - 1

- The write operations executed by all processors are seen by all processors in the order that they were issued.
- For any memory position all processors see all write operations in the same order.
- Generally considered a fairly good mind model of real world multiprocessors.

This is clearly weaker than SC since the second condition is only valid for write operations to a single memory cell.

The second condition does imply that all processors agree on the final value of a memory cell.

Example 1 (Allowed under PC but not SC):

```plaintext
// P1       // P2
x=1       // M1    y=1       // M3
r1=y      // M2    r2=x      // M4
```

// PC allows (as CC) r1==0,r2==0, since x,y are different memory cells
// P1: M1 M2 M3 M4    P2: M3 M4 M1 M2
// SC "only" allows (r1==1,r2==1), (r1==0,r2==1), (r1==0,r2==1)
Processor Consistency (PC) - 2

Basic notion can also be formulated as follows: Writes by a single processor are received by all other processors in the order in which they were issued, but writes from different processors may be seen in a different order. Reflects the idea of a network in which some nodes are closer.

Relation to Cache Coherence: Cache Coherence
Processor Consistency (PC) - 3

Example 2 (not allowed under PC):

```
// P1 // P2
x=1   // M1 y=1   // M4
c=1   // M2 c=2   // M5
r1=y   // M3 r2=x   // M6
```

// PC does not allow the result (r1==0, r2==0),
// since it follows that
// P1 has seen M1 M2 M3 M4 M5 M6
// P2 has seen M4 M5 M6 M1 M2 M3
// this means P1 and P2 do not agree about the
// order of the write operations to c

This is allowed in Weak Consistency!
**Processor Consistency (PC) - 3**

Example (allowed in PC): (“Load bypassing Stores”)

```
// P1        // P2
f1=1       // M1    f2=1       // M5
a=1        // M2    a=2        // M6
r1=a       // M3    r3=a       // M7
r2=f2      // M4    r4=f1      // M8  // <-- both loads are pulled forward,
                        // therefore 0
// PC allows  r1==1, r3==2,  r2==0, r4==0
//             ----- -----      
// P1:         M1 M2 M3 M4 M5 M6 M7 M8
// P2:         M5 M6 M7 M8 M1 M2 M3 M4

// since r2=0, SC requires the (common) order
// P1/P2:     M1 M2 M3 M4 M5 M6 M7 M8
// but then r4==1 would follow => Not allowed in SC.
```
Weak Consistency (WC)

A very weak memory model!

- Processors may not agree on the order of memory accesses.

- Synchronization is done by the programmer using explicit sync-operations.

- For the sync-operations SC is guaranteed. The orders of operations in between synchronization points can be changed for optimization.
Real World Implementations

Remark: Intel x86/64 is “Total Lock Order + Causal Consistency (TLO+CC)”, Remark: Sparc is “Total Store Order (TSO)”, and SC ⊂ TSO ⊂ TLO+CC;

According to [4] PC ⊃ TSO, since TSO allows W → R and Read-Own-Write-Early. PC allows additionally Read-Other’s-Write-Early.

According to Sevcek et al. still a usable semantics for x86 is x86-TSO, i.e. version of TSO, total store order (aka Sparc memory model).

For most usable examples: PC ≈ TSO ≈ TLO+CC.

Difference of PC vs SC and TSO is “Read-Others-Write-Early”.

Recall:

“Read-Own-Write-Early”:
After a write a processor can complete a read to the same location even though the value has not been made visible to all other processors (i.e., the other processors will read the old value); allowed in PC, SC, TSO.

“Read-Others-Write-Early” (= non-atomic write):
A processor can read a value written by another processor before it is visible to all others; allowed in PC, not in SC, TSO.
Sparc Total Storage Order (TSO)

(SPARC Architecture Manual V8, p.5):

Total Store Ordering (TSO) applies both to uniprocessors and to shared-memory multiprocessors. The memory model guarantees that the stores, FLUSHes, and atomic load-stores of all processors are executed by memory serially in an order that conforms to the order in which the instructions were issued by processors.

All SPARC implementations must support TSO. An additional model called Partial Store Ordering (PSO) is defined, which allows higher-performance memory systems to be built.

Literature generally considers TSO an implementation of PC ("TSO aka PC"), i.e., relaxed $w \rightarrow r$ allows hiding latency of writes. There are subtle differences ("TSO ensures write atomicity where PC does not")
Goals

- You know what cache coherence is.
- You know the MESI protocol.
- You know the notions
  - Sequential Consistency (SC),
  - Processor Consistency (PC), and
  - Weak Consistency (WC),
and can give examples.
Literatur


// Beispiel fuer PC
// (aus dem Intel 64 Architecture Memory Ordering White Paper fuer TLO+CC)
//
// x==y==0
// P0 // P1 // P2
// x=1 // M1 r1=x // M2 r2=y // M4
// y=1 // M3 r3=x // M5

// M1<M2 dann (wg. M2<M3) auch M1<M3;
// dann r1==1, r2==1 und r3==1;
//
// alternativ M2<M3<M1<M4<M5
// r1==0, r2==1; r3==1
//
// alternativ M4<M5<M2<M3<M1
// dann r1==0, r2==0; r3==0;
//
// NICHT erlaubt ist aber
// r1==1, r2==1, r3==0