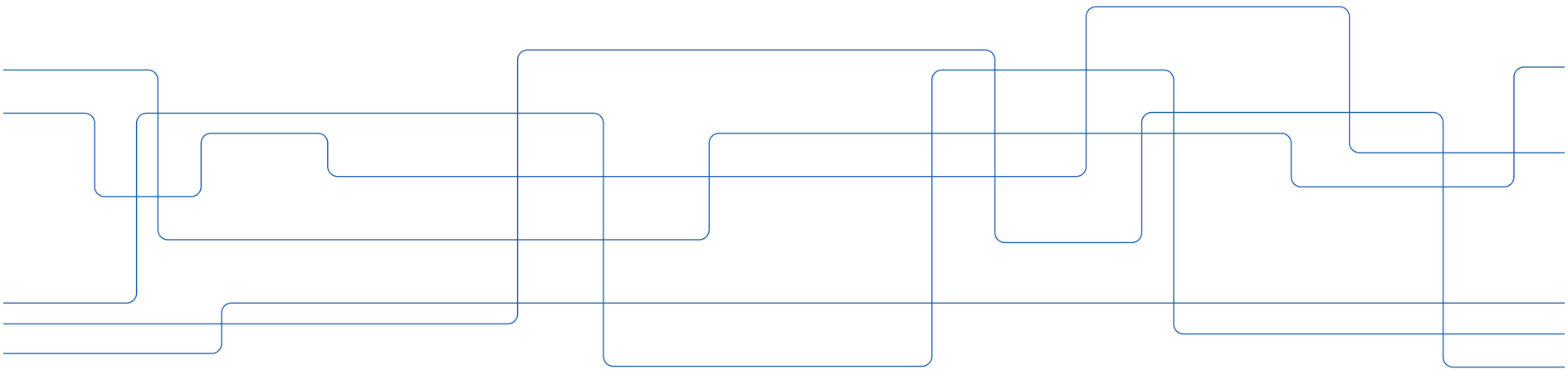

# HPC Computer Architectures: Part II

## AQTIVATE Training Workshop I

**Dirk Pleiter**

**CST | EECS | KTH**

**November 2023**



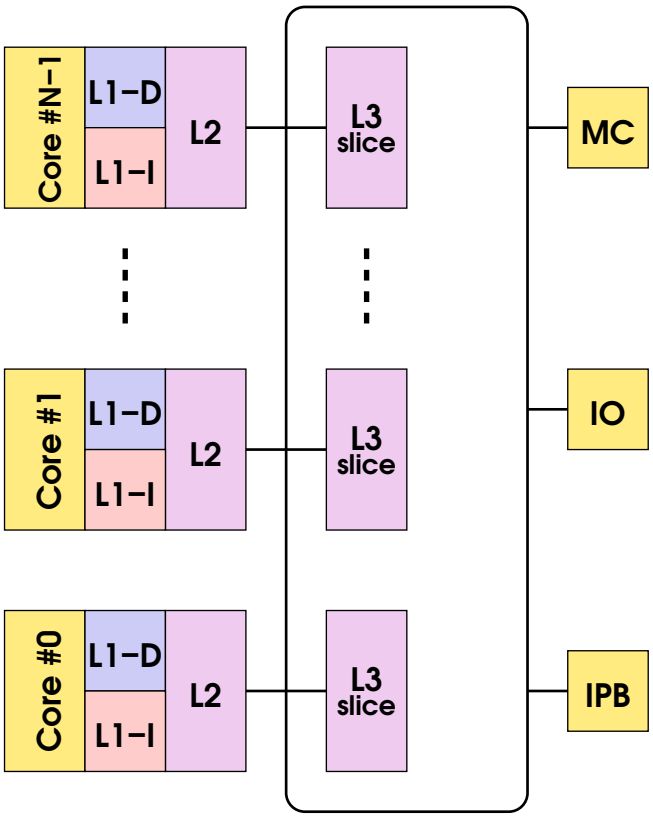
# Overview

Processor Core Architecture (1)

Memory Architecture (1)

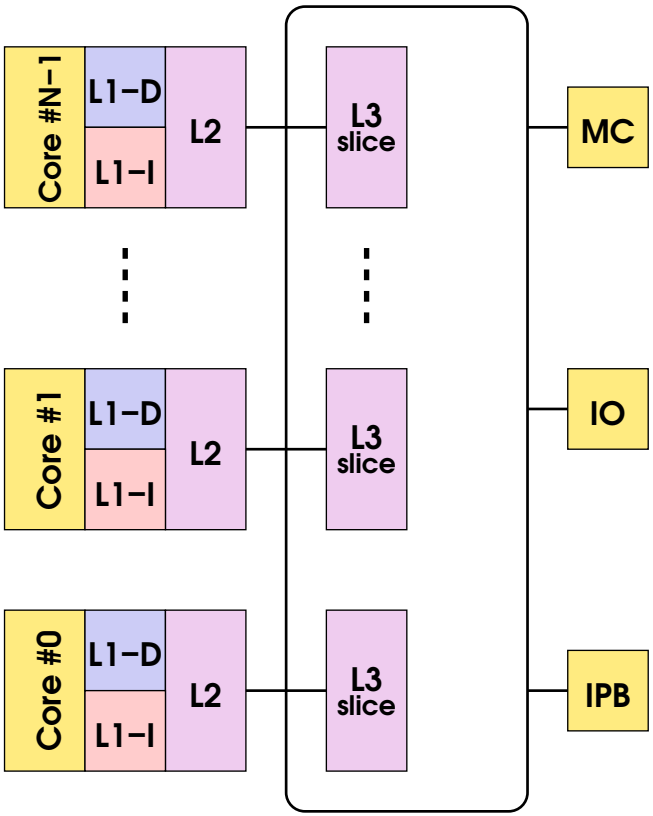

# Content

# Processor Architecture: Core




- Multi-core architectures have become the default
- Typical components
  - Front-end
    - Instruction fetching
    - Instruction decoding
    - Branch prediction
  - Execution engine
    - Resource allocation
    - Instruction execution
    - Instruction retirement

# Processor Architecture: Other




- Memory hierarchy
  - Caches
  - Memory controller
- I/O controller
  - Interface to I/O devices
  - Example: network controller
- Inter-Processor Bus
  - Required for multi-socket designs, only
- Intra-processor network
  - Ring or mesh network
  - Cross-bar switch

# Instruction Set Architecture (ISA)

- Relevant ISA categories
  - **CISC**: Complex Instruction Set Computer
  - **RISC**: Reduced Instruction Set Computer
  - **VLIW**: Very Long Instruction Word
- Classification based on internal memory architecture
  - Today's architectures are typically **general-purpose register architectures**
  - **Load-store architectures**
    - Explicit load-store operations only, operands use register addresses only
    - Examples: POWER ISA, ARM ISA
  - **Register-memory architecture**
    - Operand may be in memory
    - Examples: x86 ISA

# ISA: Operators

Selected set of categories for instruction operators:

| Operator type | Examples |
|---|---|
| Data transfer | Load and store operations |
| Arithmetic and logic | Integer arithmetics, bitwise operations, compare operations |
| Control | Branch, jump, procedure call and return |
| Floating point | Floating point arithmetics |
| Mathematical functions | Inverse, square root |

# ISA: Memory addressing

Classification according to memory addressing features

- Memory addressing
  - Granularity: Today typically byte addressing
  - Alignment requirements
- Addressing modes (`R9` ... destination register):

| Mode | Pseudo-assembler | Description |
|---|---|---|
| Register | `mov R0, R9` | Value from registers |
| Immediate | `mov 0x3, R9` | Constant values |
| Register indirect | `ld (R1), R9` | Value from location referenced by pointer |
| Displacement | `ld 0x100(R1), R9` | Pointer plus a fixed offset |
| Indexed | `ld (R0+R1), R9` | Pointer plus run-time offset |
| Scaled | `ld 0x100(R0,2), R9` | Pointer plus scaled run-time offset |

# Processor Core Performance

- Program latency (= **time-to-solution**) may be written as

$$\Delta t = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Options to reduce time-to-solution
  - Reduce number of instructions
  - Decrease **cycles per instruction (CPI)**
  - Increase clock rate

# Optimising Pipelined Architectures

- **CPI** = Cycles per Instruction
- Assume a pipelined architecture:

$$\begin{aligned}\text{Pipeline CPI} &= \text{Ideal pipeline CPI} + \\ &\quad \text{Structural stalls} + \text{Data hazard stalls} + \\ &\quad \text{Control stalls}\end{aligned}$$

  - **Structural stalls** = Resource conflicts in functional units of a pipeline
  - Possible causes for structural stalls
    - Functional units with different throughput $B$
    - Memory bus which is used for data and instructions
      ☞ conflicting load operations
- CPI can be reduced by reducing ideal pipeline CPI or stalls


# Instruction Level Parallelism (ILP)

- **ILP** = Parallelism among instructions from small code areas which are independent of one another
- Exploitation of ILP
  - Overlapping of instructions in a pipeline
  - Parallel execution of instructions in multiple functional units

# ILP: Pipeline

- Consider previous example: $z = a \times b + c - d$

| | | |
|---|---|---|
| 1 | `mul R0, R1,` **R5** | store $a \times b$ in register 5 |
| 2 | | stall |
| 3 | | stall |
| 4 | | stall |
| 5 | `add` **R5**`, R2,` **R6** | add $c$ |
| 6-8 | ... | stall |
| 9 | `sub` **R6**`, R3, R7` | subtract $d$ |

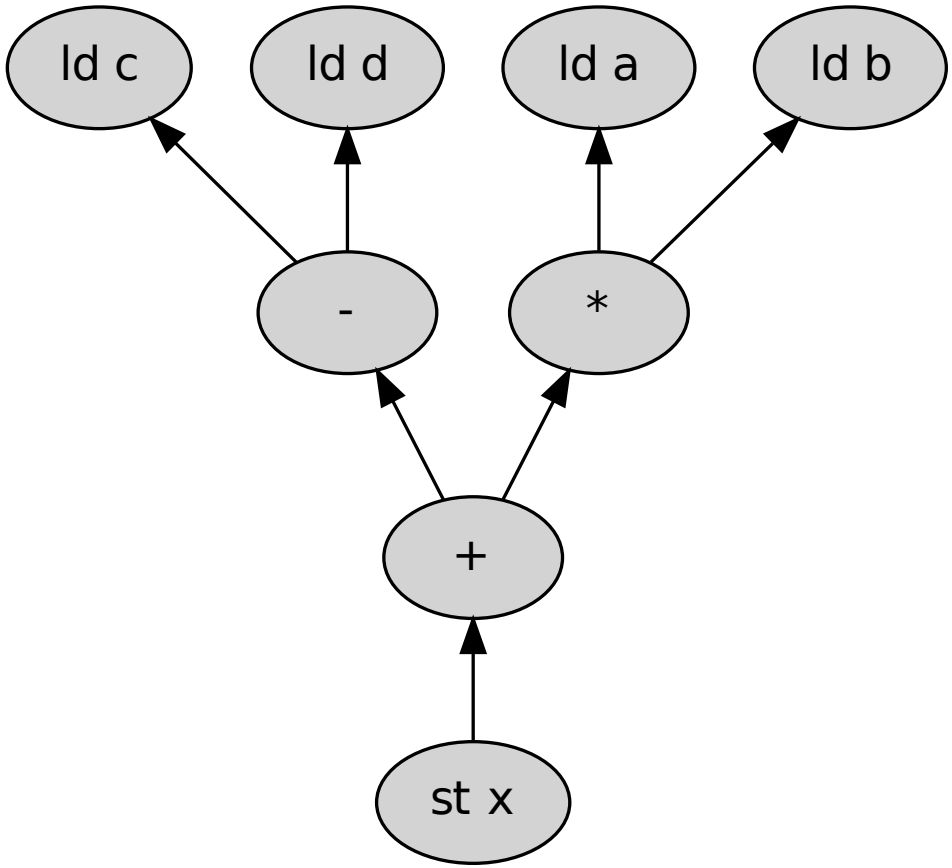
- Re-organisation to improve pipeline filling

| | | |
|---|---|---|
| 1 | `mul R0, R1,` **R5** | store $a \times b$ in register 5 |
| 2 | `sub R2, R3,` **R6** | store $c - d$ in register 6 |
| 3 | | stall |
| 4 | | stall |
| 5 | | stall |
| 6 | `add` **R5**`,` **R6**`, R7` | add $a \times b$ and $c - d$ |

# ILP: Pipeline (2)

Data dependence diagram for re-ordered schedule

Previously omitted load-store operations included



# ILP: Multiple-issue of Instructions

- Allow multiple instructions to be issued within same clock cycle
- Flavours of multiple-issue processors:
  - **Very Long Instruction Word (VLIW) processors**
    - Fixed number of instructions are scheduled per clock cycle
    - Static scheduling at compile time
  - **Statically scheduled superscalar processors**
    - **Superscalar processor** = Processor which can schedule $\geq 1$ instructions per clock cycle
    - Variable number of instructions are scheduled per clock cycle and executed **in-order**
  - **Dynamically scheduled superscalar processors**
    - Variable number of instructions are scheduled per clock cycle and executed **out-of-order**

# SIMD Parallelism

- Single Instruction Multiple Data (SIMD) instructions exploit data-level parallelism by operating on data items in parallel
  - E.g., SIMD add

$$\begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} \leftarrow \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

- Example ISA:
  - Intel Streaming SIMD Extensions (SSE)
  - Intel Advanced Vector Extensions (AVX, AVX2, AVX512)
  - POWER ISA (VMX/AltiVec, VSX)
  - ARMv7 NEON, ARMv8, ARM SVE

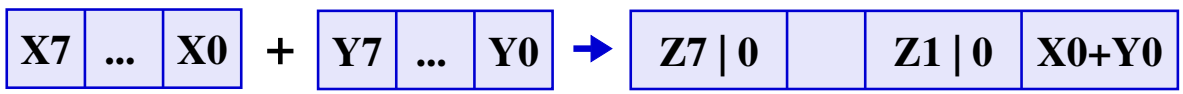
# AVX512 ISA: Example

```c
#include <stdio.h>
#include <immintrin.h>

typedef union { __m512d v; double d[8]; } Pdouble;

int main(void) {
  int i;
  Pdouble a, b, c, d;

  for (i = 0; i < 8; i++) {
    a.d[i] = b.d[i] = (double) i + 0.5;
  }

  c.v = _mm512_add_pd(a.v, b.v);

  printf("c = (%.2f,%.2f)\n", c.d[1], c.d[0]);

  return 0;
}
```

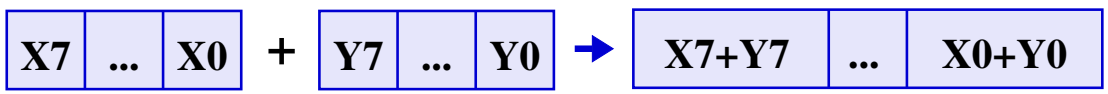
Compile with: `gcc -mavx512f ...`

# AVX512 ISA: Floating Point Instructions

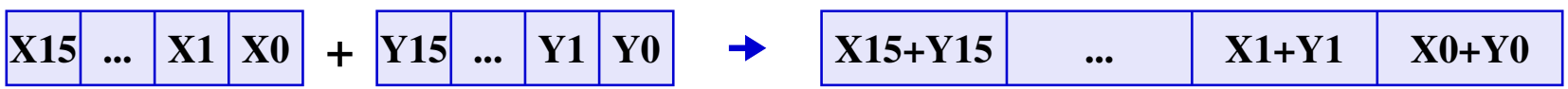
- In the following me assume all mask bits set to '1'
- Floating point instructions operate on
  - Single (S) or double (D) precision operands
  - Scalar (S) or packed (P) operands
- Scalar operations update only lowest part of destination register
  - `VADDSD`:




- Packed operations update all parts
  - `VADDPD`:

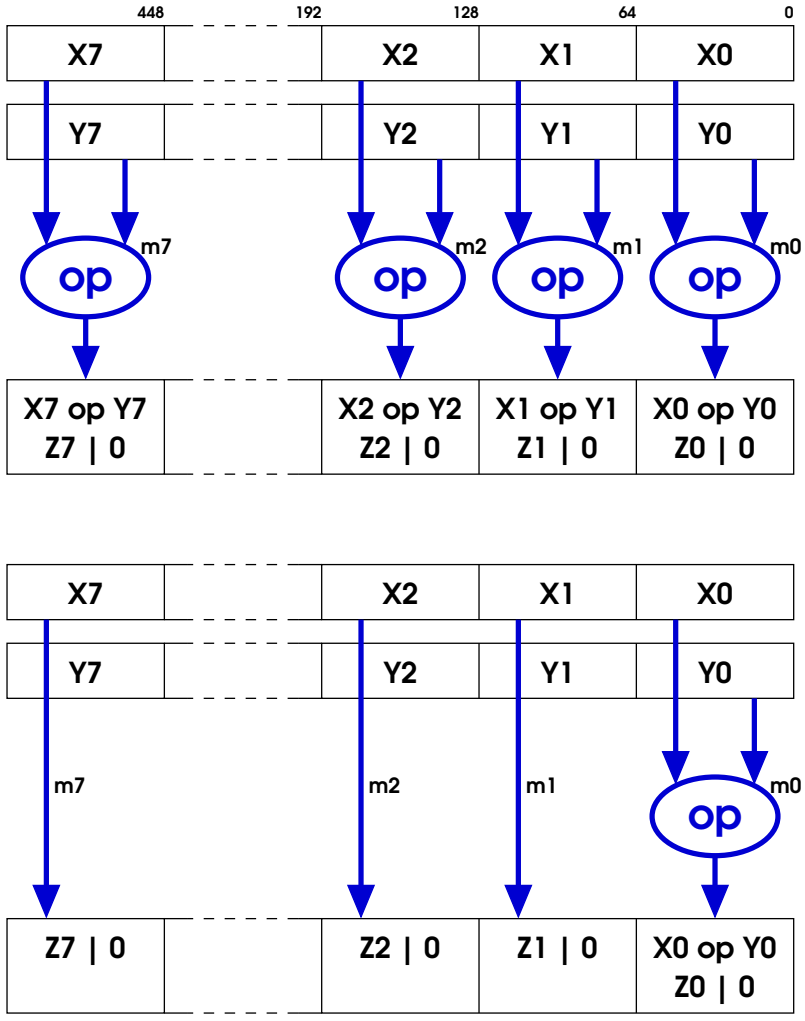



  - `VADDPS`:



# AVX512 ISA

- 512-bit wide vector units processing packed or scalar operands
  - 16× 32-bit
  - 8× 64-bit
- Support of masks: merging (m) and zeroing (z) masking
- Vector registers `zmm0`, `xmm1`, ..., `zmm31`
- 3-4 operands
  - One source can be a memory reference
  - Destination can be a memory reference



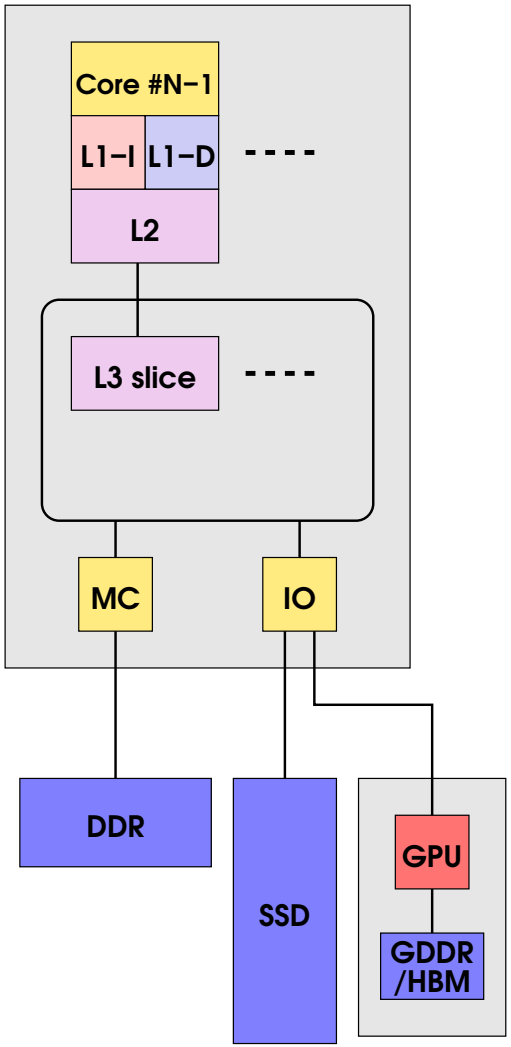
# AVX512 ISA (2)

- Types of instructions
  - Load/store instructions
  - Arithmetic and bitwise instructions
  - Shuffle and unpack instructions
  - Memory prefetch instructions
- Multiple sets of AVX512 instructions: F, CD, ER, PF, ...
  - On Linux see `/proc/cpuinfo` to identify supported sets

# Content

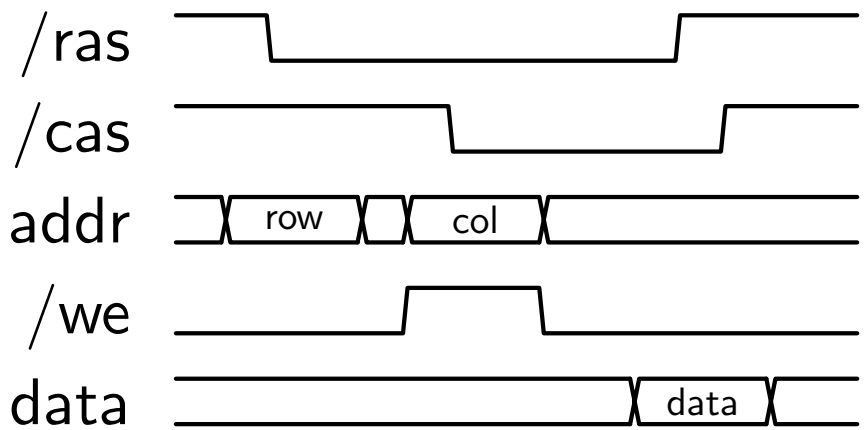
# Memory in Modern Node Architectures

- **Today's compute nodes: different memory types and technologies**
  - **Volatile memories**
    - Main memory (DDR3, DDR4, or DDR5)
    - Caches (SRAM)
    - Accelerator memory (GDDR or HBM)
  - **Non-volatile memories**
    - SSD (NAND flash, 3D XPoint)
- **Different capabilities**
  - Bandwidth
  - Latency for single transfer
    - Access time
    - Cycle time
  - Capacity



# Memory: DRAM

- Data bit storage cell = 1 capacitor + 1 transistor
- Capacitors discharge ☞ periodic **refresh** needed
  - Refresh required in intervals of O(10) ms
  - Possibly significant performance impact
  - Memory access latencies become variable
- Memory organisation
  - Memory arrays with separate **row** and **column** addresses
    ☞ less address pins
  - Read operation:



# Memory: DRAM (2)

- Technology improvements
  - Synchronization with system bus: Synchronous dynamic random-access memory (SDRAM)
  - Double Data Rate (DDR)
- DRAM typically packaged on small boards:
  **DIMM** = Dual Inline Memory Modules
  - Typical DRAM channel width: 8 Byte
- Performance range for selected set of data rates:

| | DDR3 | DDR4 | DDR5 |
|---|---|---|---|
| Standards release | 2007 | 2012 | 2020 |
| Data rate [Gbps] | 0.4 – 2.333 | 2.133 – 3.2 | 4 – 8 |
| GByte/sec/DIMM [GByte/s] | 3.2 – 18.6 | 17.1 – 25.6 | 32 – 64 |

# Memory: SRAM

- Data bit storage cell = (4 + 2) transistors
- Comparison with DRAM

| | DRAM | SRAM |
|---|---|---|
| Refresh required? | yes | no |
| Speed | slower | faster |
| Density | higher | lower |

- Typically used for on-chip memory

# Memory Access Locality

- Empirical observation: Programs tend to reuse data and instructions they have used recently
- Observation can be exploited to
  - Improve performance
  - Optimize use of more/less expensive memory
- Types of localities
  - **Temporal locality**: Recently accessed items are likely to be accessed in the near future
  - **Spatial locality**: Items whose addresses are near one another tend to be referenced close together in time

# Memory Access Locality: Example

```
double a[N][N], b[N][N];

for (i=0; i<N; i++)
  for (j=1; j<N; j++)
    a[i][j] = b[j-1][0] + b[j][0];
```

- Assume right-most index being fastest running index
- Temporal locality: `b[j][0]`
- Spatial locality: `a[i][j]` and `a[i][j+1]` ($j + 1 < N$)

# Memory/Storage Hierarchy

Computers provide multiple levels of volatile memory and non-volatile storage

- Important because of **Processor-memory performance gap**
- Close to processor: very fast memory, but less capacity

| Level | Access time | Capacity |
|---|---|---|
| Registers | O(0.1) ns | O(1) kByte |
| L1 cache | O(1) ns | O(10-100) kByte |
| L2 cache | O(10) ns | O(100) kByte |
| L3 cache | O(10) ns | O(1-10) MByte |
| Memory | O(100) ns | O(10-100) GByte |
| Disk | O(10) ms | O(100-...) GByte |

# Memory: Cache

**Cache miss** = Word not found in cache

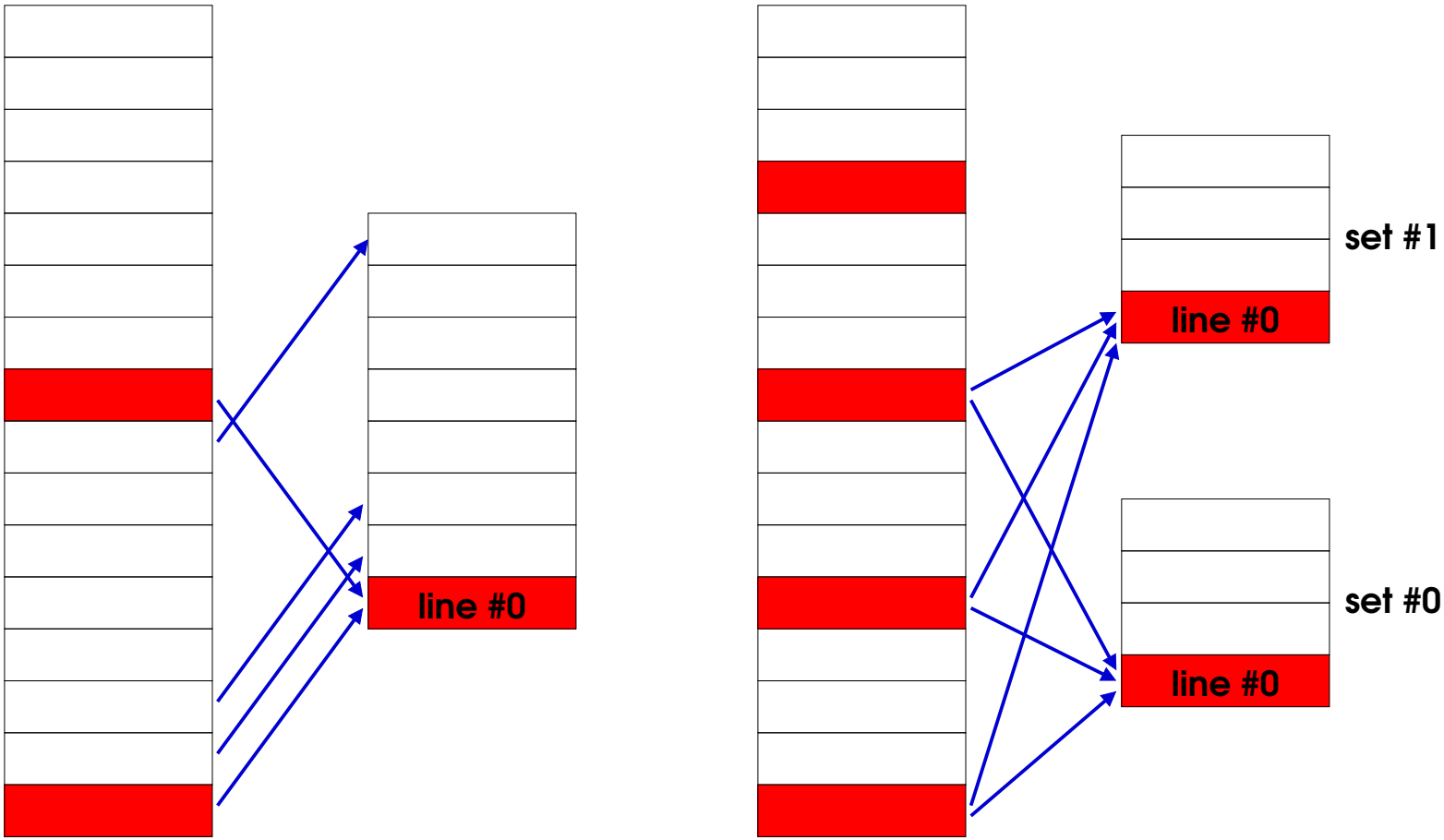
- Word has to be fetched from next memory level
- Typically a full **cache line** (=multiple words) is fetched

Cache organisation: **Set associative cache**

- Match line onto a set and then place line within the set
  - Choice of set address: (address) mod (number of sets)
- Types:
  - **Direct-mapped cache**: 1 cache line per set
    - Each line has only one place it can appear in the cache
  - **Fully associative cache**: 1 set only
    - A line can be placed anywhere in the cache
  - **n-way set associative cache**: $n$ cache lines per set
    - A line can be placed in a restricted set of places (1 out of $n$ different places) in the cache

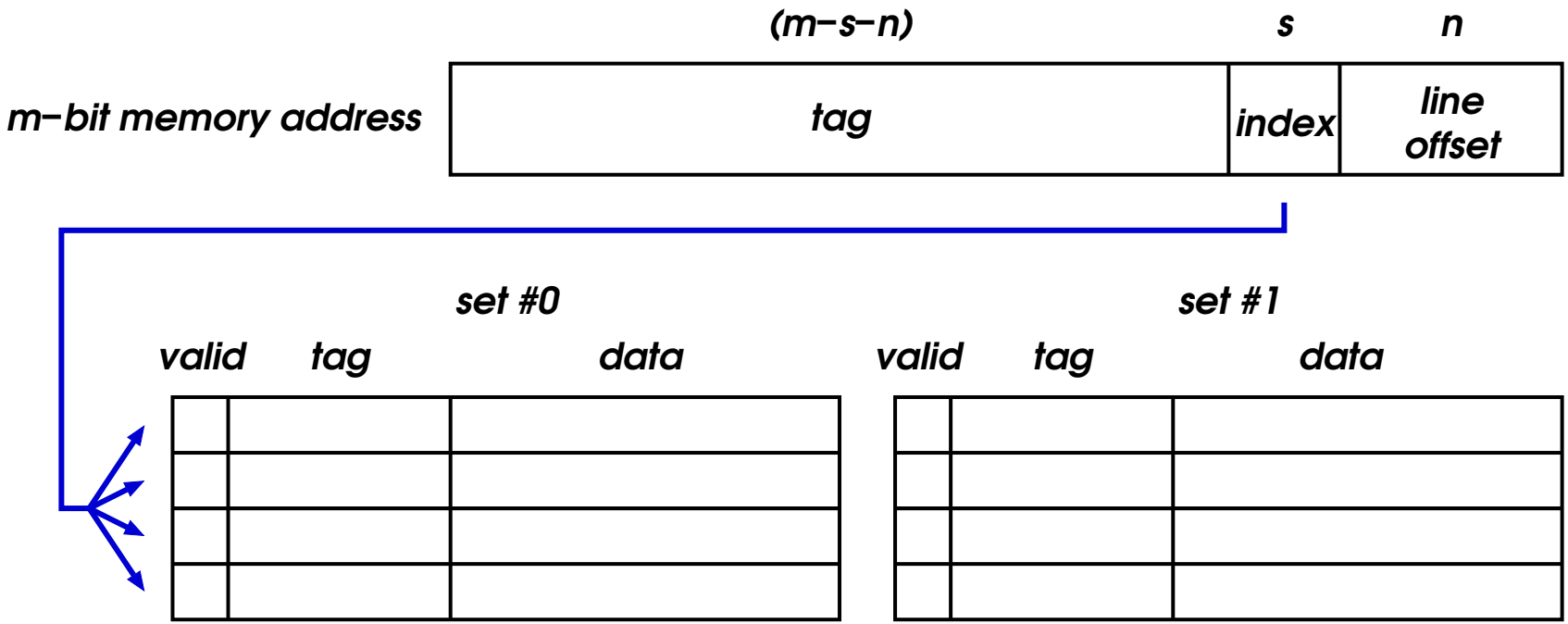
# Cache Organisation

Direct-mapped vs. 4-way set associative cache:



# Cache Addressing

Example: 4-way set associative cache with 2 sets ($s = 2$)




- Tag allows to identify set
- Set index defines location within set


# Cache Replacement Policies

- **Random**:
  - Randomly select any possible line to evict
- **Least-recently used (LRU)**:
  - Assume that a line which has not been used for some time is not going to be used in the near future
  - Requires keeping track of all read accesses
- **First in, first out (FIFO)**:
  - Book keeping simpler to implement than in case of LRU

# Cache Write Policies

Issue: How to keep cache and memory consistent?

- Cache write policy design options
  - **Write-through cache**
    - Update cache and memory
    - Memory write causes significant delay if pipeline must stall
  - **Write-back cache**
    - Update cache only and post-pone update of memory
      ☞ memory and cache will become inconsistent
    - If cache line is **dirty** then it must be committed before being replaced
    - Add dirty bit to manage this task

Issue: How to manage partial cache line updates?

- Typical solution: **write-allocate** (fetch-on-write)
  - Cache line is read before being modified

# Cache misses

- Categories of cache misses

  **Compulsory**: Occurs when cache line is accessed the first time

  **Capacity**: Re-fetch of cache lines due to lack of cache capacity

  **Conflict**: Cache line was discharged due to lack of associativity

- **Average memory access time** $=$

$$\text{hit time} + \text{miss rate} \cdot \text{miss penalty}$$

Hit time ... Time to hit in the cache

Miss penalty ... Time to load cache line from memory

# Cache misses (2)

Strategies to reduce miss rate:

- **Larger cache line size**
  - If spatial locality is large then other parts of cache line are likely to be re-used
- **Bigger caches to reduce miss rate**
  - Reduce capacity misses
- **Higher associativity**
  - Reduce conflict misses
  - Hit time may increase

# Cache misses (3)

Strategies to reduce miss penalty

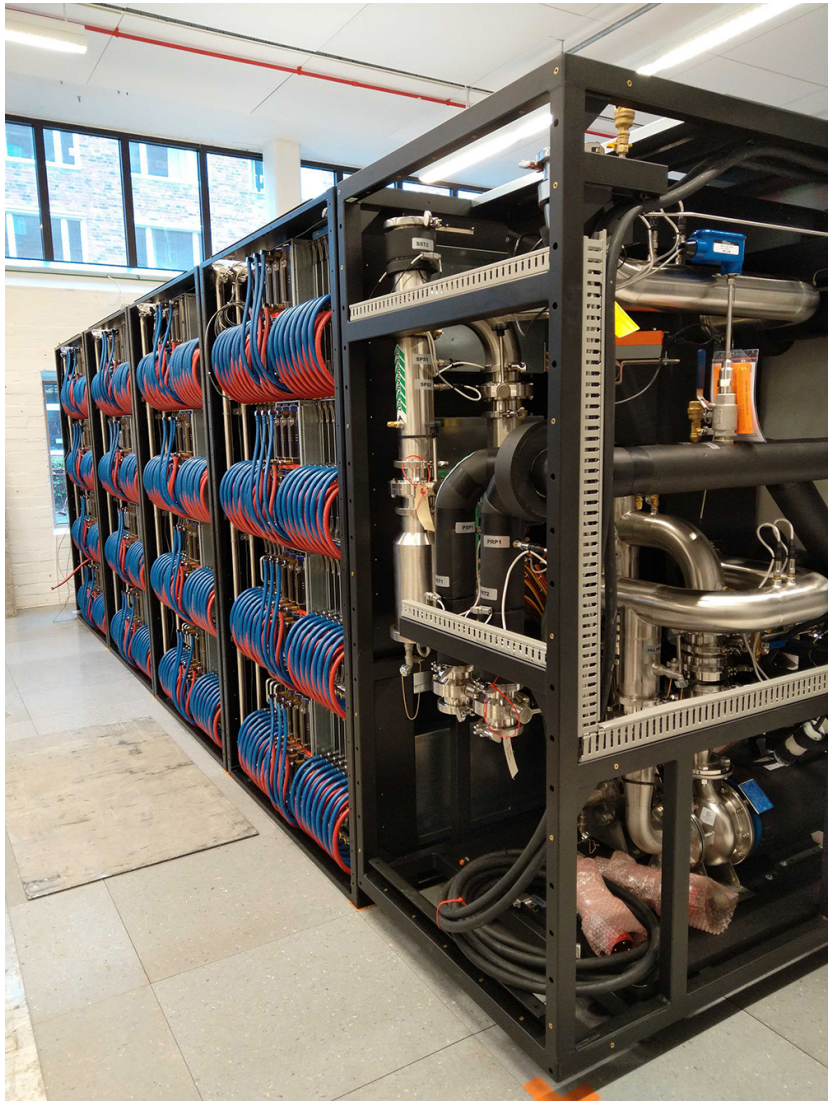
- **Multilevel caches**
  - Technical problem: Larger cache means higher miss penalty
  - Problem mitigation: Larger but slower L2 cache:

$$\text{Average memory access time} = \text{Hit time}_{\text{L1}} + \text{Miss rate}_{\text{L1}} \cdot (\text{Hit time}_{\text{L2}} + \text{Miss rate}_{\text{L2}} \cdot \text{Miss penalty}_{\text{L2}})$$

# Cache: Inclusive vs. Exclusive

- Policies:
  - **Inclusive cache** = Data present in one cache level also present in higher cache level
    - Example: All data in L1 cache is also present in L2 cache
  - **Exclusive cache** = Data present in at most one cache level
  - Intermediate policies:
    - Data available at higher cache level is allowed to be available at lower cache levels
- Advantages of inclusive cache
  - Management of cache coherence may be simpler
- Advantages of exclusive cache
  - Duplicate copies of data in several cache levels avoided
- Example for intermediate policy:
  - L3 cache which is used as **victim cache**: cache holds data evicted from L2, but not necessarily data loaded to L2

# Finish with an Architecture at PDC: Dardel



[PDC, 2021]