Introduction to Modern Processor Architecture

Jan Eitzinger (RRZE)
Georg Hager (RRZE)
Milestone Inventions

- 1938 Elwood Shannon: Solve boolean algebra and binary arithmetic with arrangements of relays
- 1941 Zuse Z3 – Electromechanical computer, binary floating-point
- 1946 ENIAC – First general purpose electronic computer
- 1949 EDVAC, EDSAC – First Stored Program Computers
- 1964 IBM System/360 – Separation between computer architecture and implementation
- 1964 CDC 6600 – First Supercomputer
- 1969 IBM 360/85 – Introduction of Memory hierarchy
- Storage is separated from the Processing Unit
- Processor is controlled by instructions
- Single storage for instructions and data
- Instructions are represented as numbers
- A problem is solved by a sequence of instructions, which manipulate data in memory
Main Principle in Computer Architecture

- The stored program computer is a very flexible design
- This flexibility makes hardware optimizations difficult

Solution:

**Computer architects try to make the common case fast!**

Consequence for the programmer:

You have to learn what the processor designers assumed to be the common case
1. **Instruction execution**

This is the primary resource of the processor. All efforts in hardware design are targeted towards increasing the instruction throughput.

Instructions are the concept of "work" as seen by processor designers. Not all instructions count as "work" as seen by application developers!

Example: Adding two arrays \(A(\cdot)\) and \(B(\cdot)\)

\[
\text{do } i=1, N \\
\quad A(i) = A(i) + B(i) \\
\text{enddo}
\]

**User work:** \(N\) Flops (ADDs)

**Processor work:**
- \(\text{LOAD } r1 = A(i)\)
- \(\text{LOAD } r2 = B(i)\)
- \(\text{ADD } r1 = r1 + r2\)
- \(\text{STORE } A(i) = r1\)
- \(\text{INCREMENT } i\)
- \(\text{BRANCH } \rightarrow \text{top if } i<N\)
2. **Data transfer**

Data transfers are a consequence of instruction execution and therefore a secondary resource. Maximum bandwidth is determined by the request rate of executed instructions and technical limitations (bus width, speed).

Example: Adding two arrays \(A(:)\) and \(B(:)\)

\[
\begin{align*}
do \ i=1, \ N \\
A(i) &= A(i) + B(i)
enddo
\end{align*}
\]

Data transfers:
- 8 byte: \(\text{LOAD } r1 = A(i)\)
- 8 byte: \(\text{LOAD } r2 = B(i)\)
- 8 byte: \(\text{STORE } A(i) = r2\)

Sum: **24 byte**

Crucial question: What is the bottleneck?
- Data transfer?
- Code execution?
Computer Architects’ Assumptions

- **Instruction-level parallelism (ILP)**

  The sequential instruction stream exhibits the potential of executing several instructions in parallel and/or in different order without altering the program semantics.
  Keywords: Pipelining, out-of-order execution, superscalarity

- **Locality of access**

  Data once loaded from memory will be needed again “soon.” Adjacent data in memory will also be needed “soon.”
  Keywords: Spatial/temporal locality, cache, cache lines

- **Data parallelism**

  Data needed for computation can be loaded, processed, and stored in parallel.
  Keywords: Single Instruction Multiple Data (SIMD)
Performance metrics

- “Pure” metrics
  - **Flop/s**: floating point operations per second

\[
\text{Flop/s} = \frac{\text{Number of Floating Point Operations executed}}{\text{execution time [s]}}
\]

(relevant for many technical & scientific applications)

- **IPC**: instructions per cycle

\[
\text{IPC} = \frac{\text{Number of Instructions executed}}{\text{execution time [cy]}}
\]

(relevant for computer architects – in practice, CPI = 1/IPC is used, too)

- “Best metric”: 1/(wallclock time)
  - Measures time to solution → best metric thinkable, but not intuitive in all situations (see later)
Performance metrics

- **Flop/s** and **IPC** numbers can be used as metrics for hardware (peak performance) and application programs (sustained performance)
- **Caveat**: Both metrics are very easy to manipulate and can be misleading!
- Standard benchmark programs
  - **LINPACK** (cf. discussion about TOP500; [http://www.top500.org](http://www.top500.org))
    LINPACK ~ Peak Performance in most cases [MFlop/s]
    STREAM ~ Performance of main memory data access [Mbyte/s]
  - **SPEC** (Standard Performance Evaluation Corporation, [http://www.spec.org](http://www.spec.org))
    A lot of different benchmarks: Web Server, Mail Server,…
    **SPEC CPU**: numerical performance
    (a collection of application benchmarks in FORTRAN or C)
  - **HPC Challenge**: Several specific HPC benchmarks
1. Increase **performance / throughput of CPU core**
   a) Reduce cycle time, i.e. increase clock speed (Moore)
   b) Increase throughput, i.e. superscalar + SIMD

2. Improve **data access time**
   a) Increase cache size
   b) Improve main memory access (bandwidth & latency)

3. Use parallel computing (**shared memory**)
   a) Requires shared-memory parallel programming
   b) Shared/separate caches
   c) Possible memory access bottlenecks

4. Use parallel computing (**distributed memory**)
   “Cluster” of computers tightly connected
   a) Almost unlimited scaling of memory and performance
   b) Distributed-memory parallel programming
How to build faster computers

*Increase single processor performance: Moore’s law*

- In 1965 G. Moore claimed
  - # of transistors on a chip doubles every 12 or 24 months (cost “sweet spot”)

- Processor speed grew roughly at the same rate
  - This is an entirely nontrivial effect!

- **Problem:** Power dissipation

This trend is currently changing: see multi-core
How to build faster computers

Faster CPUs may not compute faster! DRAM Gap!

General computer architecture
Basic features of modern microprocessors

How do we make use of all these transistors?

- Pipelining
- Superscalarity
- Single Instruction Multiple Data (SIMD) processing
- Simultaneous Multi-Threaded processing (SMT)
The abstraction stack of programming

- **Application**: High Level Programming Language (e.g. C / C++ / Fortran) – portable

- **Compiler** translates program to machine-specific machine instructions (IA32, IA64)

- Modern computers/microprocessors – von Neumann concept is still visible, but
  - Several memory levels (3-4)
  - Multiple arithmetic/logical units (e.g. 8 hardware units for integer and FP operations on Itanium2)
Simple view of modern processors

Cache based microprocessors (e.g. Intel, AMD)

Processor components:

- Arithmetic & functional units, e.g. FP multiply, FP add, integer, MMX,…
- These units can only use operands resident in the registers
- Operands are read (written) by load (store) units from main memory/caches to registers
- Caches are fast but small pieces of memory (5-10 times faster than main memory)
- A lot of additional logic: e.g. branch prediction, reorder buffer, TLBs
 Benchmarking data transfers:  
*The Vector Triad*

```fortran
double precision, dimension(N) :: A,B,C,D  
A=1.d0; B=A; C=A; D=A  
stime = timestamp()  
do j=1,NITER  
  do i=1,N  
    A(i) = B(i) + C(i) * D(i)  
  enddo  
  if(.something.that.is.never.true.) then  
    call dummy(A,B,C,D)  
  endif  
enddo  
etime = timestamp()  
Mflops = (2.d0*NITER)*N / (etime-stime) / 1.d6
```

- Report performance for different N, choose NITER so that accurate time measurement is possible
- This kernel is limited by data transfer performance for all memory levels on all architectures, ever!

Prevent smarty-pants compilers from doing “clever” stuff
Serial vector triad performance

- Vector triad “feels” the location of the data in the memory hierarchy

- The closer the data to the registers, the faster the code

- VT is purely data bound on all existing architectures

- What about multiple cores? → see later!

\[
A(\cdot) = B(\cdot) + C(\cdot) \times D(\cdot)
\]
The “big four” of modern core architecture

**Pipelining:**
Instruction execution in multiple steps

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fetch Instruction 1 from L1I</td>
</tr>
<tr>
<td>2</td>
<td>Decode Instruction 1</td>
</tr>
<tr>
<td>3</td>
<td>Execute Instruction 1</td>
</tr>
<tr>
<td>4</td>
<td>Fetch Instruction 2 from L1I</td>
</tr>
<tr>
<td>5</td>
<td>Decode Instruction 2</td>
</tr>
<tr>
<td></td>
<td>Execute Instruction 2</td>
</tr>
<tr>
<td></td>
<td>Fetch Instruction 3 from L1I</td>
</tr>
<tr>
<td></td>
<td>Decode Instruction 3</td>
</tr>
<tr>
<td></td>
<td>Execute Instruction 3</td>
</tr>
<tr>
<td></td>
<td>Fetch Instruction 4 from L1I</td>
</tr>
<tr>
<td></td>
<td>Decode Instruction 4</td>
</tr>
<tr>
<td></td>
<td>Execute Instruction 4</td>
</tr>
<tr>
<td></td>
<td>Fetch Instruction 5 from L1I</td>
</tr>
<tr>
<td></td>
<td>Decode Instruction 5</td>
</tr>
<tr>
<td></td>
<td>Execute Instruction 5</td>
</tr>
<tr>
<td></td>
<td>Fetch Instruction 6 from L1I</td>
</tr>
<tr>
<td></td>
<td>Decode Instruction 6</td>
</tr>
<tr>
<td></td>
<td>Execute Instruction 6</td>
</tr>
</tbody>
</table>

**Superscalarity:**
Multiple instructions per cycle

**Single Instruction Multiple Data:**
Multiple operations per instruction

**Simultaneous Multi-Threading:**
Multiple instruction sequences in parallel
Split complex instructions (e.g., multiplication) into several simple / fast sub-operations (stages)

Makes short cycle time possible (simpler logic circuits), e.g.:
- Multiplication takes 5 cycles ("latency"), but
- processor can work on 5 different multiplications simultaneously
- Can produce one result each cycle after the pipeline is full ("throughput" = 1 cycle)

Drawbacks:
- Pipeline must be filled - startup times
- Requires complex instruction scheduling by compiler/hardware → software pipelining / out-of-order
- Extensive use requires large number of independent instructions – instruction level parallelism

Vector supercomputers use multiple parallel pipelines with SIMD execution
- And they have the memory bandwidth to feed them!
Pipelining:
5-stage Multiplication-Pipeline: \( A(i) = B(i) \times C(i) ; i=1,\ldots,N \)

<table>
<thead>
<tr>
<th>Cycle:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>...</th>
<th>N+4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Separate Mant. / Exp.</td>
<td>B(1) C(1)</td>
<td>B(2) C(2)</td>
<td>B(3) C(3)</td>
<td>B(4) C(4)</td>
<td>B(5) C(5)</td>
<td>B(6) C(6)</td>
<td>...</td>
</tr>
<tr>
<td>Mult. Mantissa</td>
<td>Mult. Mantissa</td>
<td>B(1) C(1)</td>
<td>B(2) C(2)</td>
<td>B(3) C(3)</td>
<td>B(4) C(4)</td>
<td>B(5) C(5)</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Add. Exponents</td>
<td>Add. Exponents</td>
<td>B(1) C(1)</td>
<td>B(2) C(2)</td>
<td>B(3) C(3)</td>
<td>B(4) C(4)</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Normal. Result</td>
<td>Normal. Result</td>
<td>A(1)</td>
<td>A(2)</td>
<td>B(3) C(3)</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insert Sign</td>
<td>Insert Sign</td>
<td>A(1)</td>
<td>A(2)</td>
<td>A(2)</td>
<td>...</td>
<td>A(N)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

First result is available after 5 cycles (= latency of pipeline)!
### Pipelining versus purely sequential execution of multiplication

<table>
<thead>
<tr>
<th>Sequential:</th>
<th>Pipelining:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 multiplication = 5 cycles</td>
<td>Start-Up = 5 cycles</td>
</tr>
<tr>
<td>N multiplications:</td>
<td>N multiplications:</td>
</tr>
<tr>
<td>$T_{\text{seq}}(N) = (5\times N)$ cycles</td>
<td>$T_{\text{pipe}}(N) = (4+N)$ cycles</td>
</tr>
</tbody>
</table>

**Speedup:**

\[
T_{\text{seq}} / T_{\text{pipe}} = \frac{(5\times N)}{(N+4)} = 5/(1 + 4/N) \sim 5 \text{ for large } N \ (>>5)
\]

**Throughput (Results per Cycle) of Pipeline:**

\[
N / T_{\text{pipe}}(N) = N / (4 + N) = 1 / (1 + 4/N) \sim 1 \text{ for large } N
\]
Pipelining
Benefits and drawbacks

- In general (m-stage pipe /pipeline depth: m)

  Speed-Up:
  \[ \frac{T_{seq}}{T_{pipe}} = \frac{(m*N)}{(N+m-1)} \sim m \text{ for large } N \ (>>m) \]

  Throughput (Results per Cycle):
  \[ \frac{N}{T_{pipe}(N)} = \frac{N}{(N+m-1)} = \frac{1}{\left[ 1+(m-1)/N \right]} \sim 1 \text{ for large } N \]

- Number of independent operations \( (N_C) \) required to achieve \( T_p \) results per cycle:

  \[ T_p = \frac{1}{\left[ 1+(m-1)/N_C \right]} \quad \Rightarrow \quad N_C = T_p \frac{(m-1)}{(1- T_p)} \]

  \[ T_p = 0.5 \quad \Rightarrow \quad N_C = m-1 \]
### Pipelining

#### Benefits and drawbacks

- **Drawbacks:**
  - N small (e.g., \(N=1\)) \(\rightarrow\) no speedup!
  - Increasing clock frequency \(\rightarrow\) pipeline depth \(m\) increases (pipeline stages must be smaller/simpler)
  - Operations (e.g. Multiplications) within pipeline must be independent!
    - If they are not, we get “bubbles”
  - Optimal scheduling of instructions by compiler/hardware depends on pipeline depth!
  - Effective pipeline length for execution of an arithmetic operation is much longer than number of pipeline stages of arithmetic unit (see below).
Basic types of (potential) dependencies within loop body may prevent efficient pipelining, e.g.:

**No dependency:**

\[
\text{do } i=1,N \\
\quad a(i) = a(i) \times c \\
\text{end do}
\]

**Dependency:**

\[
\text{do } i=2,N \\
\quad a(i) = a(i-1) \times c \\
\text{end do}
\]

**Pseudo-dependency:**

\[
\text{do } i=1,N-1 \\
\quad a(i) = a(i+1) \times c \\
\text{end do}
\]
Efficient pipelining
Or lack thereof…

Concurrent updates in one loop

2300 Mflop/s = 1 Flop/cy

460 Mflop/s = 1 Flop / 5 cy

“Haswell” 2.3 GHz
(SIMD turned off in compiler)
Pipelining

Further potential problems

- Typical number of pipeline stages: 3-6 for the hardware pipelines on modern CPUs (FP MULT/ADD)
- 1 or 2 MULT/ADD units per processor core (or 1-2 FMA units)
- Modern microprocessors have limited pipelining for div / sqrt and no hardware units for exp / sin / …!

**Example**: Cycles per Operation (DP) on Xeon “Sandy Bridge”

<table>
<thead>
<tr>
<th>Operation</th>
<th>y=a+y (y=a*y)</th>
<th>y=a/y</th>
<th>y=sqrt(y)</th>
<th>y=exp(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>3 (5)</td>
<td>≤ 23</td>
<td>≤ 23</td>
<td>20-200</td>
</tr>
<tr>
<td>Throughput</td>
<td>1 (1)</td>
<td>≤ 22</td>
<td>≤ 21</td>
<td>20-200</td>
</tr>
<tr>
<td>Cycles/Operation</td>
<td>0.25*</td>
<td>≤ 11*</td>
<td>≤ 10.5*</td>
<td>5-50*</td>
</tr>
</tbody>
</table>

- Reduce number of complex operations if possible
- Replace function call with a table lookup if the function is frequently computed for a few different arguments only.

* If using packed AVX instructions (see later)
Besides the arithmetic and functional units, the instruction execution itself is pipelined also, e.g.: one instruction performs at least 3 steps:

fetch instruction from L1I → decode instruction → execute instruction

Hardware Pipelining on processor (all units can run concurrently):

1. Fetch Instruction 1 from L1I
2. Fetch Instruction 2 from L1I → Decode Instruction 1
3. Fetch Instruction 3 from L1I → Decode Instruction 2 → Execute Instruction 1
4. Fetch Instruction 4 from L1I → Decode Instruction 3 → Execute Instruction 2

... 

- Branches can stall this pipeline! (speculative execution, predication)
- Each unit is pipelined itself (cf. Execute=MULT Pipeline)
Superscalar Processors

- Superscalar Processors can run multiple instruction pipelines at the same time!

- Parallel hardware components / pipelines are available to:
  - fetch / decode / issue / retire multiple instructions per cycle (typically 2-6)
  - load (store) multiple operands (results) from (to) cache per cycle (typically 1-3 full registers per cycle)
  - perform multiple integer / address calculations per cycle (e.g., 2-3 address generation units [AGUs] on Intel CPUs)
  - Execute multiple floating point instructions per cycle (typically 2)

- On superscalar RISC processors out-of order execution hardware is available to optimize the usage of the parallel hardware
Superscalar Processors
Instruction Level Parallelism through superscalar execution

- Multiple units enable use of Instruction Level Parallelism (ILP):

- Issuing m concurrent instructions per cycle: m-way superscalar
- Modern processors are 3- to 6-way superscalar (with 1-2 FP instructions per cycle)
A further way to increase performance: SIMD

- Pipelining is a way to leverage parallelism on the instruction level
- Superscalar processors leverage parallelism across pipelines
- A third way is to introduce parallelism on the data level such that every instruction does more work

**SIMD** = “single instruction multiple data”

Instructions operate on **wide registers** (128-512 bits), which hold multiple data items

→ one instruction amounts to several operations!
**SIMD**

- **Advantage:** If you use these instructions you can benefit
  - Most performance benefit for data in registers or L1 cache
- **Disadvantage:** No data parallelism → no SIMD
  - Ideally, the compiler recognizes the data parallelism in loops automatically and uses appropriate SIMD instructions
  - Data layout may be decisive!

- **Examples**
  - x86: SSE* / AVX / AVX2 / AVX512
  - Intel Xeon Phi Knights Landing: AVX512
  - IBM Power: AltiVec/VSX
  - ARM: Neon/SVE
**Core details: SIMD processing**

- Single Instruction Multiple Data (SIMD) instructions allow the **concurrent execution** of the same operation on “wide” registers.
- Adding two registers holding double precision FP operands:

\[
\begin{align*}
A[0] & + B[0] & \rightarrow C[0] \\
\end{align*}
\]

**SIMD execution:** 
\[V64ADD \ [R0,R1] \rightarrow R2\]

**Scalar execution:** 
\[R2 \leftarrow \text{ADD} \ [R0,R1] \]
SIMD processing:
Structure of arrays (SoA) data layout

real, dimension(():) :: A,B,C,D,R
SIMD processing:
Array of structures (AoS) data layout

```
type struct
  real*4  A, B, C, D, R
end type struct

type(struct), dimension(:) :: S
```
SIMD processing – Basics

Steps (done by the compiler) for "SIMD processing"

for(int i=0; i<n;i++)
    C[i]=A[i]+B[i];

“Loop unrolling”

for(int i=0; i<n;i+=4){
    C[i] =A[i] +B[i];
    C[i+1]=A[i+1]+B[i+1];
}

//remainder loop handling

LABEL1:
VLOAD R0 ← A[i]
VLOAD R1 ← B[i]
V64ADD[R0,R1] → R2
VSTORE R2 → C[i]
i←i+4
i<(n-4)? JMP LABEL1

//remainder loop handling

Load 256 Bits starting from address of A[i] to register R0
Add the corresponding 64 Bit entries in R0 and R1 and store the 4 results to R2
Store R2 (256 Bit) to address starting at C[i]

Do not unroll your code explicitly for “better SIMD”!
**SIMD processing – Basics**

- **No SIMD vectorization** for loops with data dependencies:

  ```c
  for(int i=0; i<n;i++)
    A[i]=A[i-1]*s;
  ```

- **“Pointer aliasing”** may prevent **SIMDfication**

  ```c
  void scale_shift(double *A, double *B, double *C, int n) {
    for(int i=0; i<n; ++i)
      C[i] = A[i] + B[i];
  }
  ```

- **C/C++** allows that \(A \rightarrow &C[-1]\) and \(B \rightarrow &C[-2]\) \(\rightarrow C[i] = C[i-1] + C[i-2]:\) dependency \(\rightarrow\) No SIMD

- **If “pointer aliasing” is not used**, tell the compiler:
  - `-fno-alias` (Intel), `-Msafeptr` (PGI), `-fargument-noalias` (gcc)
  - `restrict` keyword (C only!):
    ```c
    void f(double restrict *a, double restrict *b) {...}
    ```
Rules for vectorizable loops

1. Countable
2. Inner loop
3. Single entry and single exit
4. Straight line code
5. No loop-carried dependencies
6. No function calls (exception intrinsic math functions)

Better performance with:
1. Simple loops with unit stride
2. Minimize indirect addressing
3. Aligned data structures (getting less important)

Obstacles for vectorization:
1. Non-contiguous memory access
2. Data dependencies
Pipelining does **not work perfectly**. This introduces bubbles in the pipeline.

**SMT duplicates** certain parts of the hardware in order to utilize these idle phases in the ALUs and keep them busy all the time.

- Must execute multiple independent instruction streams

This is advantageous also to **hide data access latencies**

There is no benefit if **shared resources** are the bottleneck

Currently implemented on

- All “higher end” Intel CPUs (2-way)
- IBM Power (8-way)
- Intel Xeon Phi KNL (4-way)
- AMD Zen and Zen2 (2-way)
- ARM ThunderX2 (4-way)

**SMT is a strictly in-core feature** to improve pipeline utilization!
SMT principle (2-way example):
SMT impact

- **SMT is primarily suited for increasing processor throughput**
  - With multiple threads/processes running concurrently

- **Scientific codes tend to utilize chip resources quite well**
  - Standard optimizations (loop fusion, blocking, …)
  - High data and instruction-level parallelism
  - Exceptions do exist

- **SMT is an important topology issue**
  - SMT threads share almost all core resources
    - Pipelines, caches, data paths
  - Affinity matters!
  - If SMT is not needed
    - pin threads to physical cores
    - or switch it off via BIOS etc.
SMT impact

- SMT adds **another layer of topology** (inside the physical core)
- Caveat: SMT threads **share all caches**!
- Possible benefit: **Better pipeline throughput**
  - Filling otherwise unused pipelines
  - Filling pipeline bubbles with other thread’s executing instructions:
    
    Thread 0:
    ```
    do i=1,N
    a(i) = a(i-1)*c
    enddo
    ```
    
    Thread 1:
    ```
    do i=1,N
    b(i) = s*b(i-2)+d
    enddo
    ```

  - **Beware**: Executing it all in a single thread (if possible) may reach the same goal without SMT:
    ```
    do i=1,N
    a(i) = a(i-1)*c
    b(i) = s*b(i-2)+d
    enddo
    ```

Dependency → pipeline stalls until previous MULT is over

Unrelated work in other thread can fill the pipeline bubbles
Simultaneous recursive updates with SMT

**Intel Sandy Bridge (desktop) 4-core; 3.5 GHz; SMT**

**MULT Pipeline depth: 5 stages → 1 F / 5 cycles for recursive update**

Fill bubbles via:
- SMT
- Multiple streams

**Thread 0:**
```
do i=1,N
   A(i)=A(i-1)*c
   B(i)=B(i-1)*d
endo
```

**Thread 1:**
```
do i=1,N
   A(i)=A(i-1)*c
   B(i)=B(i-1)*d
endo
```

**MULT pipe**
Simultaneous recursive updates with SMT

Intel Sandy Bridge (desktop) 4-core; 3.5 GHz; SMT
MULT Pipeline depth: 5 stages → 1 F / 5 cycles for recursive update

5 independent updates on a single thread do the same job!
Memory hierarchy
Bridging the DRAM Gap:
Memory hierarchies

Old-school vector processor
- Main memory
- Vector registers
- Arithmetic units

Cache-based microprocessor
- Main memory
- Floating Point registers
- L1 Cache
- L2 Cache
- L3 Cache

Application
- Manipulation/Computation

Arithmetic units

Floating Point registers

"DRAM Gap"

Data
Two quantities characterize the quality of each memory hierarchy (main memory & caches):

- **Latency** $T_{\text{lat}}$: *Time to set up the memory transfer from source (main memory or caches) to destination (registers).*

  Latency is usually given in ns (memory & off-chip caches) or processor cycles (on-chip caches)

- **Bandwidth** $\text{BW}$: *Maximum amount of data which can be transferred per second / cycle between source (main memory or caches) and destination (registers).*

  Bandwidth is usually given in GByte/s (memory & off-chip caches) or DP floating point words (double precision) per processor cycle (on-chip caches)

**Overall data transfer time:**

$$T = T_{\text{lat}} + \frac{\text{(amount of data)}}{\text{BW}}$$
### Memory Hierarchies Today: Intel x86

**Intel Xeon Haswell-EP**

<table>
<thead>
<tr>
<th>Single core specs</th>
<th>Peak perf. Clock freq.</th>
<th># FP Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>41.6 GFlop/s 2.6 GHz</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Level</th>
<th>Size</th>
<th>BW</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 D</td>
<td>32 KB</td>
<td>&gt;200 GB/s</td>
<td>4 cycles</td>
</tr>
<tr>
<td>L2</td>
<td>256 KB</td>
<td>~80 GB/s</td>
<td>10-12 cycles</td>
</tr>
<tr>
<td>L3</td>
<td>up to 45 MB (shared)</td>
<td>up to 700 GB/s</td>
<td>~30 cycles</td>
</tr>
<tr>
<td>Mem.</td>
<td>Socket BW</td>
<td>~60 GB/s (measured)</td>
<td>~150 cycles</td>
</tr>
</tbody>
</table>

**2014**

Intel Haswell: up to 18 cores

![Diagram of memory hierarchy](image)
Solutions to the latency problem

- Hide latencies by overlapping load and execute stages
- Prefetch data explicitly from memory ahead of its use in computation
- Interleave memory to hide bank busy time

Solutions to bandwidth problem

- Provide several layers of memory with different sizes and bandwidths
  - Level 1,2 cache: fast (runs at CPU speed), low latency; can saturate many CPU operations; on-chip cache
  - Level 3,4 cache: slower, somewhat larger latency; sometimes runs at lower frequency; sometimes off-chip
  - Main memory: slowest, large latency, interleaved

- Problem: the faster, the more expensive, the smaller!
- Large CPU-speed L2 or L3 caches are now common
Memory hierarchies: Caches and their organization
Memory hierarchies

Example: Write-allocate write-back cache

How does data travel from memory to the CPU and back?

- Caches are organized in cache lines (e.g., 64 bytes)
- Only complete cache lines are transferred between memory hierarchy levels (except registers)
- MISS: Load or store instruction does not find the data in a cache level → CL transfer required

- Example: Array copy $A(:) = C(:)$
Memory Hierarchies: 

**Cache Structure**

- Cache line fetch/load has **large latency penalty**
- “Neighboring” items can then be used from cache

```plaintext
do i=1,n  
s = s+a(i)*a(i)
enddo
```

**Cache line size:** 4 **words**

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LD, Cache miss : Latency, Use data</td>
</tr>
<tr>
<td>2</td>
<td>LD, Use data</td>
</tr>
<tr>
<td>3</td>
<td>LD, Use data</td>
</tr>
<tr>
<td>4</td>
<td>LD, Use data</td>
</tr>
<tr>
<td>5</td>
<td>LD, Cache miss : Latency</td>
</tr>
<tr>
<td>6</td>
<td>Use data</td>
</tr>
<tr>
<td>7</td>
<td>LD, Use data</td>
</tr>
<tr>
<td>8</td>
<td>LD, Use data</td>
</tr>
</tbody>
</table>
Memory Hierarchies: 
*Cache Structure*

- Cache line data is **always consecutive**
  - Cache use is **optimal for contiguous access** ("stride 1")
  - Non-consecutive access reduces performance
  - Access with **large stride** (e.g. with cache line size) can lead to disastrous **performance breakdown**

- **Long cache lines** reduce the latency problem if only contiguous memory access is done. Otherwise latency problem becomes worse
  - Tradeoff between STREAM bandwidth and application bandwidth

- Calculations get cache bandwidth inside the cache line, but main memory latency still limits performance

- Cache lines must somehow be mapped to memory locations
  - Cache **multi-associativity** enhances utilization
  - Try to avoid **cache thrashing**
Memory hierarchies: Hiding memory latency by prefetching
Prefetch (PFT) instructions:
- Transfer of consecutive data (one cache line) from memory to cache
- Followed by LD to registers
- Useful for executing loops with consecutive memory access
- Compiler has to ensure correct placement of PFT instructions
  - Knowledge about memory latencies required
  - Loop timing must be known to compiler
- Due to large latencies, a minimal number of outstanding prefetches must be sustained

Many architectures (Intel/AMD x86, Power4/5/6/7) have hardware-based automatic prefetch mechanisms
- HW detects regular patterns (streams) and prefetches at will
- Intel x86: “Adjacent cache line prefetch” loads 2 (64-byte) cache lines on L3 miss → Effectively doubles line length on loads
Memory Hierarchies: Cache Line Prefetch

Array operation with prefetch:

→ Overlap of data transfer and calculation!

```
\begin{align*}
\text{do } &i=1,n \\
& \quad s = s + a(i) \times a(i) \\
\text{enddo}
\end{align*}
```

Two outstanding prefetches required to bridge latency
Characterization of Memory Hierarchies

- Relating bandwidth to peak performance: balance of a processor

  **Machine Balance:** $(\text{BW [GByte/s]} \) / \text{Peak Performance [GFlop/s]}$

  Amount of input data that can be delivered by the memory hierarchy for each available FP operation
  → This number depends on the memory hierarchy level!

- Modern processors: 0.1 byte/Flop to 0.5 byte/Flop (main memory)

- Balance can also be determined for application programs (loops):

  **Code Balance:** $(\# \text{ bytes transferred}) / (\# \text{ FP operations})$

  → This number may also depend on the memory hierarchy level (see later)
  → basic input for loop performance modeling (see later)
Evolution of machine balance

Machine balance $B_{mem}$ [byte/flop]

NHL  Nehalem EP
SNB  Sandy Bridge EP
HSW  Haswell EP
BDW  Broadwell EP
SKL  Skylake SP
KNC  Knights Corner
KNL  Knights Landing

NEC SX-8 (4 B/F)
NEC SX-9 (2.56 B/F)
NEC SX-ACE
NEC Tsubasa


Year

single core  multicore

(C) 2020 RRZE, LRZ

General computer architecture
Processor design is optimized to make “the common case” fast
The clock speed race has ended in ≈ 2003 (power dissipation!)
Moore’s Law still gives us more transistors every year
  • But the end of Moore’s Law is already in sight! What’s next???
Higher performance on the chip is achieved by
  • Faster instruction execution per cycle (pipelining, SMT, superscalarity)
  • Wider SIMD width, i.e. more operations per instruction (SSE → AVX → AVX512)
  • More cores on the socket
    ➢ Trend towards many-core computing
      (many simple cores → Xeon Phi, GPGPUs)
Memory interface bandwidth does not keep up with computation
  • Caches can alleviate this in some cases
Cache properties are important to know about
  • Cache lines and prefetching alleviate latency effect
  • Understanding performance == knowing where the data goes
  • Avoiding slow data paths is the key to good performance