HOCHSCHULE ESSLINGEN

LECTURE COMPUTER ARCHITECTURE

ADVANCED MICROPROCESSOR ARCHITECTURES: INTEL X86

RAINER KELLER

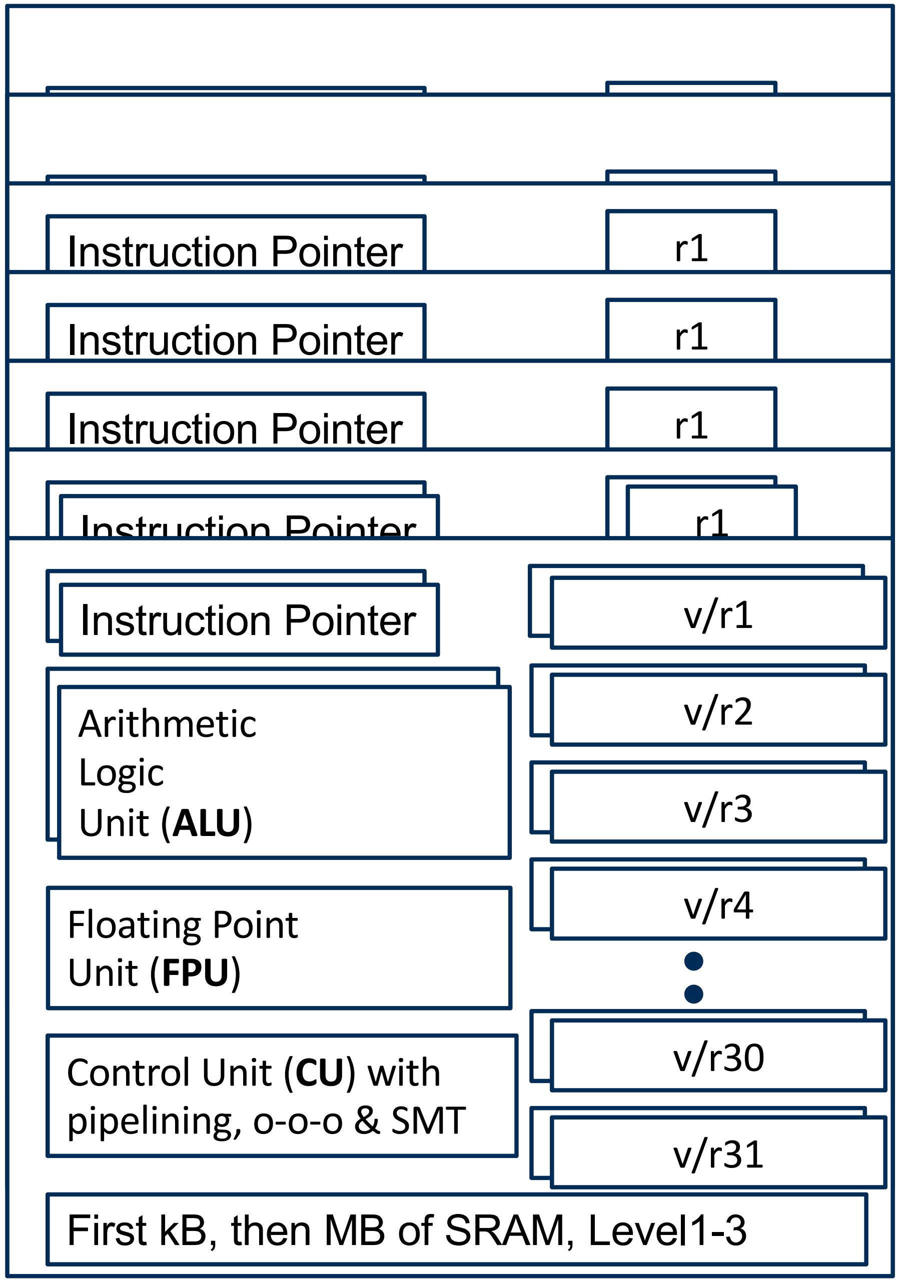
CONTENT

Advanced Topics of CPUs

REPETITION MICROPROCESSORS

Steps for improvement: Check 01_CA_*

1. Instruction Pipelining (shorter circuits → higher MHz, complex CU, from 5 → 34 stages)
2. Add integrated Cache (fast SRAM decouples DRAM, may be I-/D-Cache, multiple levels
3. Multiple Instructions per Cycle (duplicate units in ALU, or complete ALU and FPU)
4. Speculative Out-of-Order Execution (compensate for bad instruction order and possibly add more register-files!)
5. Hyperthreading / SMT (not 1 thread of execution, but have HW track 2 threads; needs duplicated state, i.e. 2 IPs, two register files, etc., complex CU)
6. MMX, SSE, AVX (Single-Instruction-Multiple Data, SIMD): vector-operations, e.g. Intel AVX-512: 8 FP doubles, 16 FP floats per register…
7. multi-core…. **Many**-core (not shown here)


Instruction Pointer
r1
Instruction Pointer
r1
Instruction Pointer
r1
Instruction Pointer
r1
Instruction Pointer
v/r1
Arithmetic Logic Unit (ALU)
v/r2
v/r3
Floating Point Unit (FPU)
v/r4
Control Unit (CU) with pipelining, o-o-o & SMT
v/r30
v/r31
First kB, then MB of SRAM, Level1-3

HOCHSCHULE ESSLINGEN

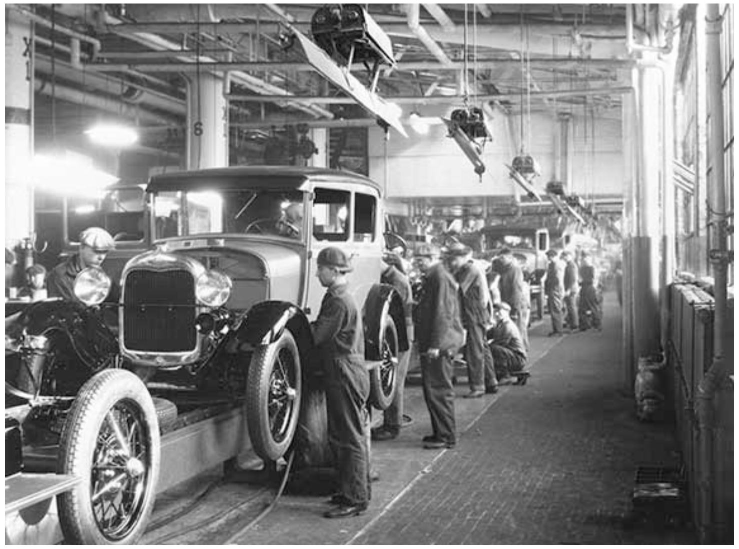
Pipelining

PIPELINING

- Pipelining visualized:

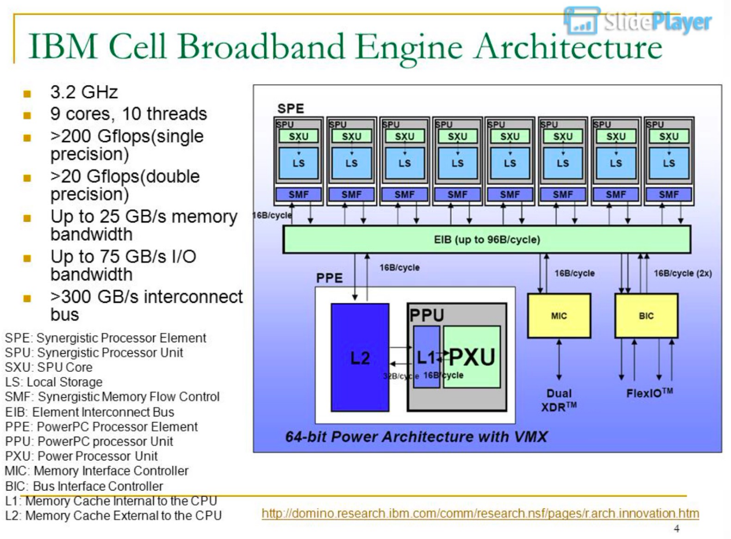
  Ford Model-T Assembly Line (1928):
  one worker repeats one assembly step.

  He’s trained to be efficient & effective.
  All complete their step at ~ same time.



Source: alf-img.com

IBM’s Cell processor with 1x PowerPC and 8 ”SPUs” in the Sony PS3:



IBM Cell Broadband Engine Architecture

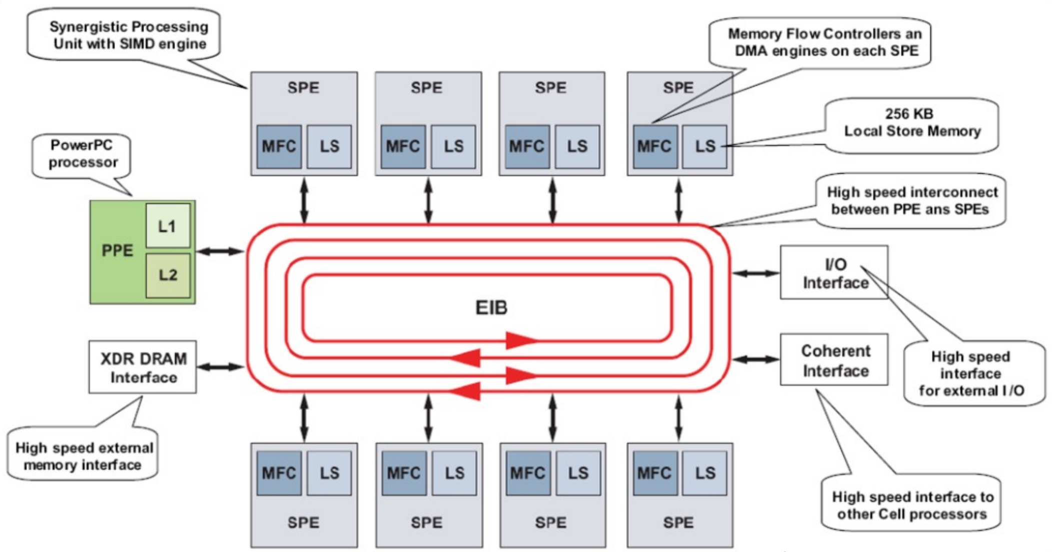
- 3.2 GHz
- 9 cores, 10 threads
- >200 Gflops(single precision)
- >20 Gflops(double precision)
- Up to 25 GB/s memory bandwidth
- Up to 75 GB/s I/O bandwidth
- >300 GB/s interconnect bus

SPE: Synergistic Processor Element
SPU: Synergistic Processor Unit
SXU: SPU Core
LS: Local Storage
SMF: Synergistic Memory Flow Control
EIB: Element Interconnect Bus
PPE: PowerPC Processor Element
PPU: PowerPC processor Unit
PXU: Power Processor Unit
MIC: Memory Interface Controller
BIC: Bus Interface Controller
L1: Memory Cache Internal to the CPU
L2: Memory Cache External to the CPU

http://domino.research.ibm.com/comm/research.nsf/pages/r.arch.innovation.htm

4

SlidePlayer
SPE
SPU
SXU
LS
SMF
16B/cycle
EIB (up to 96B/cycle)
16B/cycle
PPE
16B/cycle
16B/cycle (2x)
L2
PPU
L1
PXU
32B/cycle
16B/cycle
MIC
BIC
Dual XDR™
FlexIO™
64-bit Power Architecture with VMX


Synergistic Processing Unit with SIMD engine
Memory Flow Controllers an DMA engines on each SPE
SPE
MFC
LS
256 KB Local Store Memory
PowerPC processor
PPE
L1
L2
High speed interconnect between PPE ans SPEs
I/O Interface
EIB
XDR DRAM Interface
Coherent Interface
High speed interface for external I/O
High speed external memory interface
High speed interface to other Cell processors

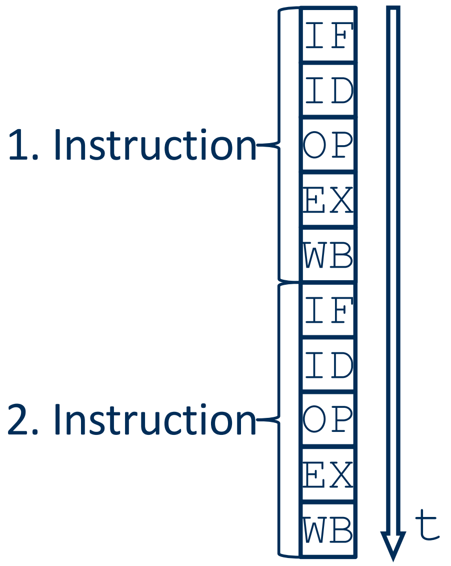
Source: SlideShare, talk Michael A. Baker

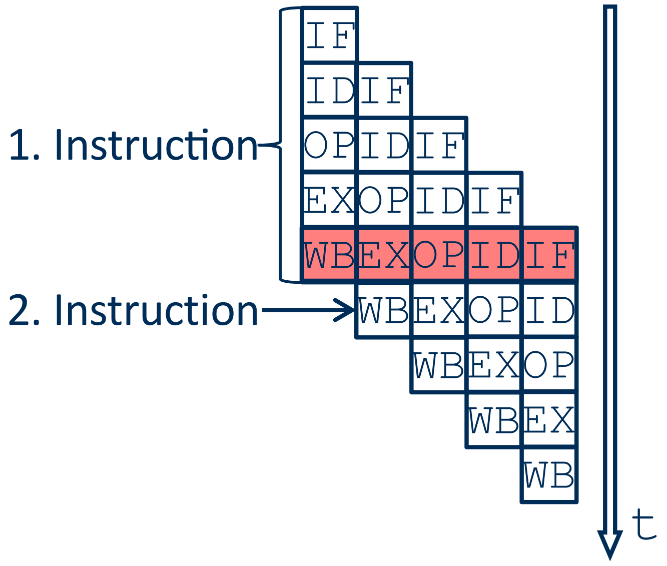
Source: redgamingtech.com / PS3 Post Mortem

PARALLELISM VIA PIPELINING

Pipelining is an important concept in many areas – not just CS!
To improve instruction execution pipelining allows the overlap of many specialized units more efficiently, **even within the CPU**!
Instructions like `add r16,0x20` may be divided into multiple smaller steps, i.e. these five phases, instead of sequential, pipeline them:

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Fetch Operands (OP)
4. Execute (EX)
5. Write Back (WB)





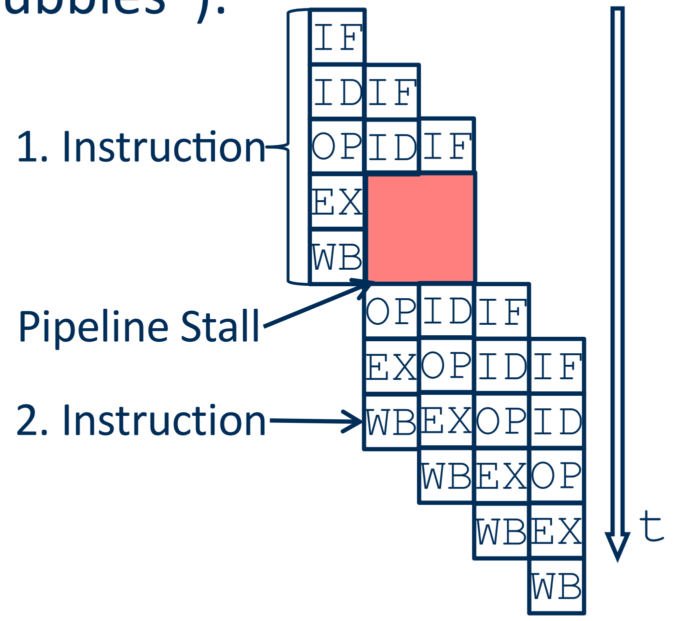
- This works only, if the instructions are **independent;** *the common case*
- If this is **not** the case, dependent instructions wait one or more cycles (so-called bubbles). In bad cases, the pipeline has to be **flushed!**

PARALLELISM USING PIPELINING 2/2

If consecutive instructions are dependent instructions; CPU has to wait one (or more) cycles (so called "pipeline stalls/bubbles"):

```
    add r16, 0x20
    mul r16, 0x2
```

The second instruction uses the same register `r16` for addition, as the multiplication instruction, which therefore has to wait…



Even worse are „unexpected" events, like

- function calls,
- (unpredicted) jumps
- hardware interrupts

leading to "pipeline flushes", aka the whole pipeline needs to be rebuilt starting from the then new processor instruction.

Today the Branch-Prediction Unit tries to eliminate unnecessary stalls! (together with Out-of-Order Execution speculatively fetches & executes)

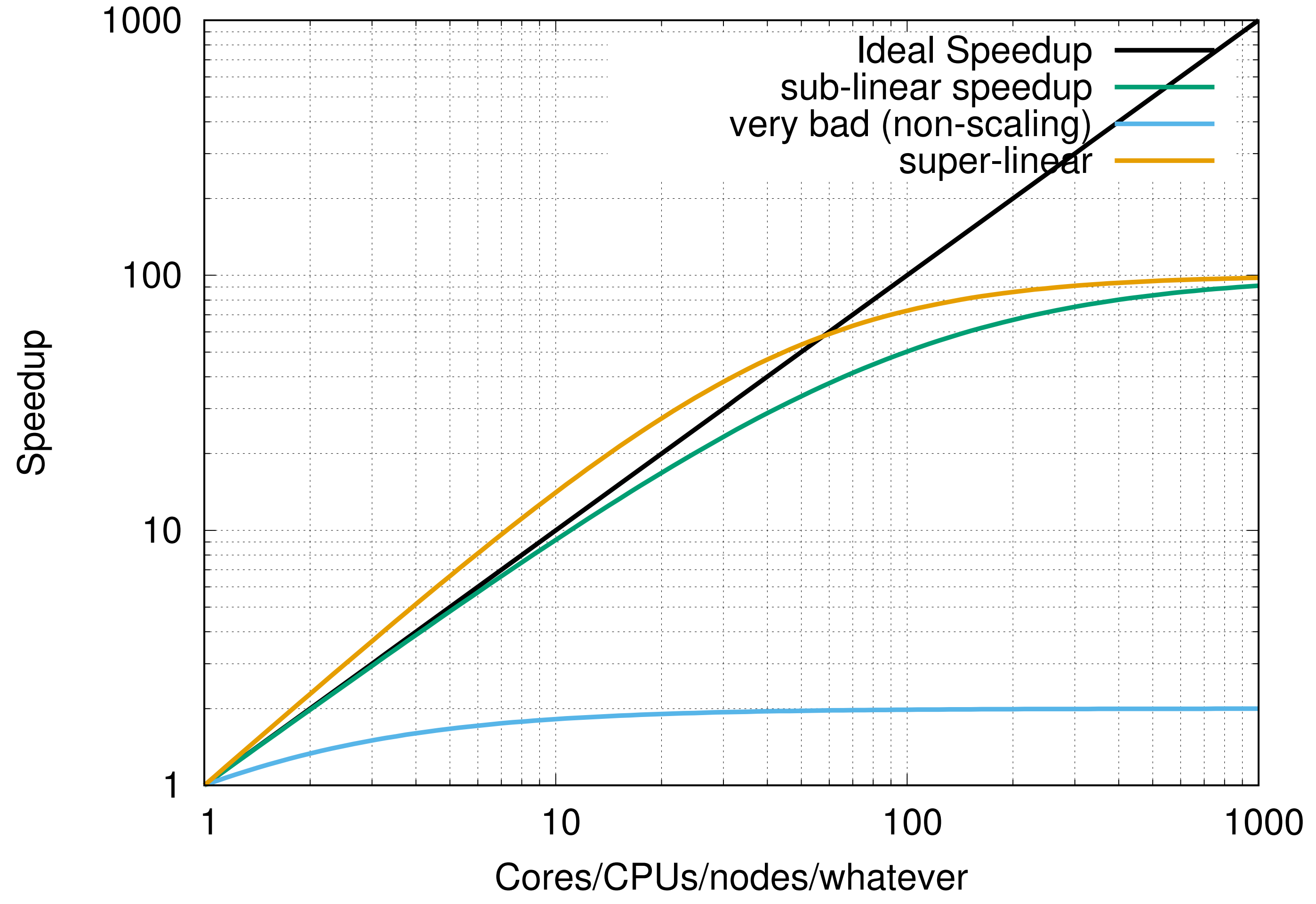
→ Leads to non-determinism…

Scaling

SCALING

HOCHSCHULE ESSLINGEN

Building construction: digging 10m ditch takes 1 person ~10hrs
How long does it take with 10 persons?



HOCHSCHULE ESSLINGEN

Caches

CACHES: SOLUTION TO AN EVER GROWING PROBLEM

A "cache" is a collection of data duplicating / shadowing data, usually to speedup the access, e.g. memory access.
A processor runs with over 3 GHz clock frequency; data has to be fetched from main memory at least once, i.e. DRAM

Access to (approximation):

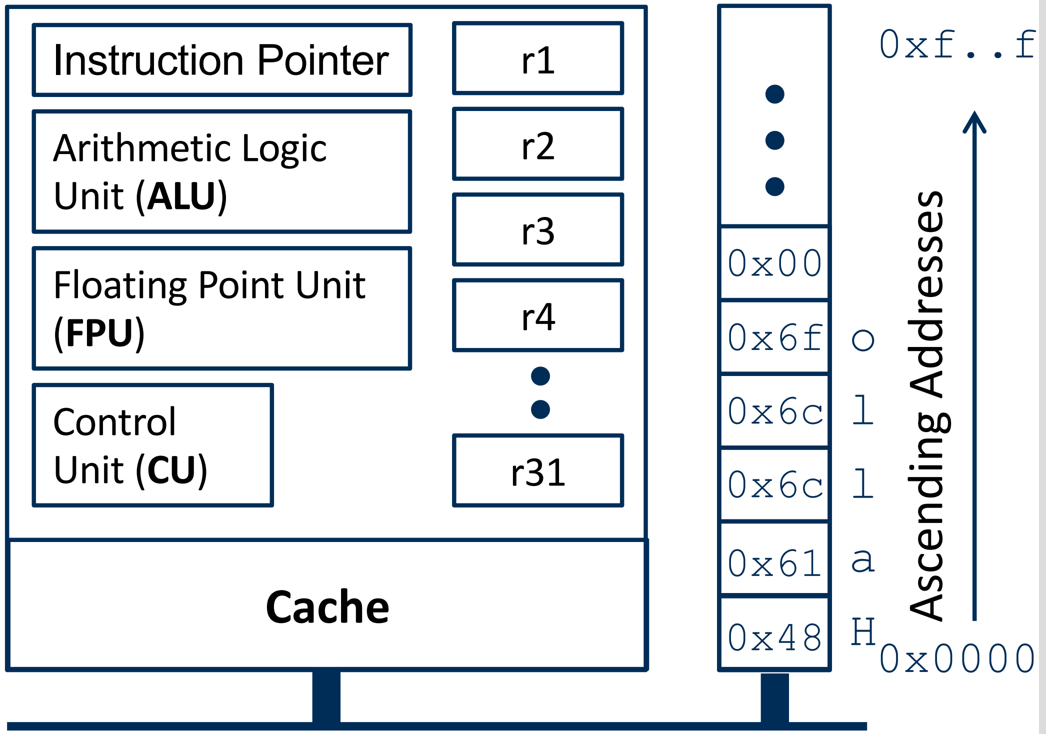
- Register: 1 cycle
- Cache: 1 - 10 cycles
- Main memory: >>100 cycles

To be fast, caches are:

- Small (1kB - few MB)
- Organized (see below)
- Tailored specifically to use-case.

Caches **only work,** if our code has:

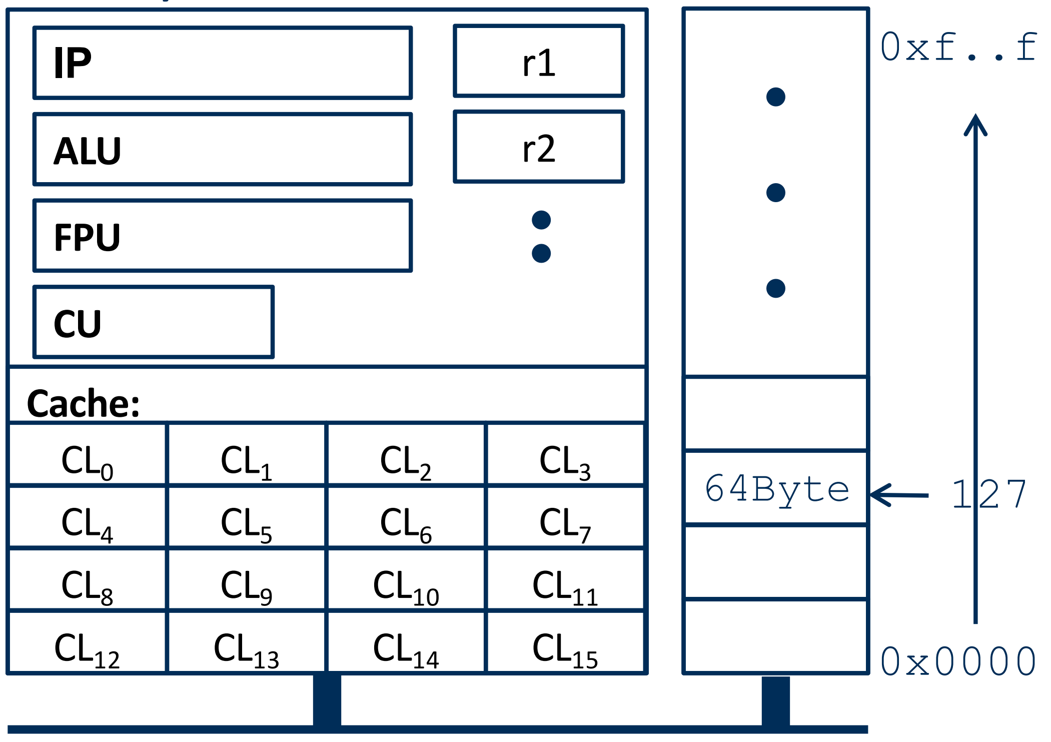
- Temporal & Spatial locality



CACHE: FUNCTION & STRUCTURE 1/2

Caches need to be fast!
For every memory access, CPU needs to see whether data is in Cache (Cache-Hit) or needs to fetch from memory (Cache-Miss)
Availability is checked using the memory accesses' **address**.

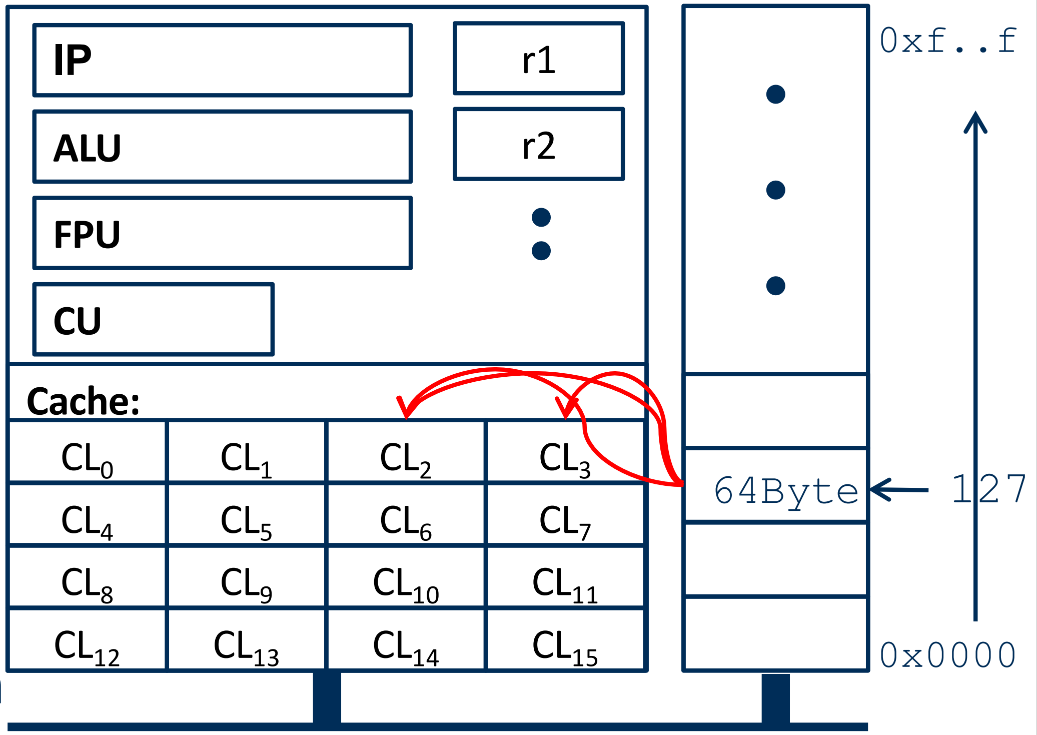
- Organized in Cache-Lines (CL) e.g. of 64 Bytes length; With a size of 1kB, there would be 16 Cache-Lines!
- Upon read from memory address `127`, every CL would have to be checked: "fully associative"
- To be practical, CLs are associated with specific memory addresses (so called Associativity) See next slides for implementations



CACHE: FUNCTION & STRUCTURE 2/2

Fully Associative (any CL may take any of the memory's blocks) are unpractical due to high power consumption to check all locations.
"One-way" associativity / Direct mapping : A memory block has only **one** possibly CL where it may be stored; not good if we have more than 16 blocks of memory…

- Two-way associativity:
  a memory block may be in one of two possibly CLs, i.e. we need two address comparators, and some eviction strategy!
- Four-way: 4 possible locations for cache lines
- …
- Fully associative **any** CL location

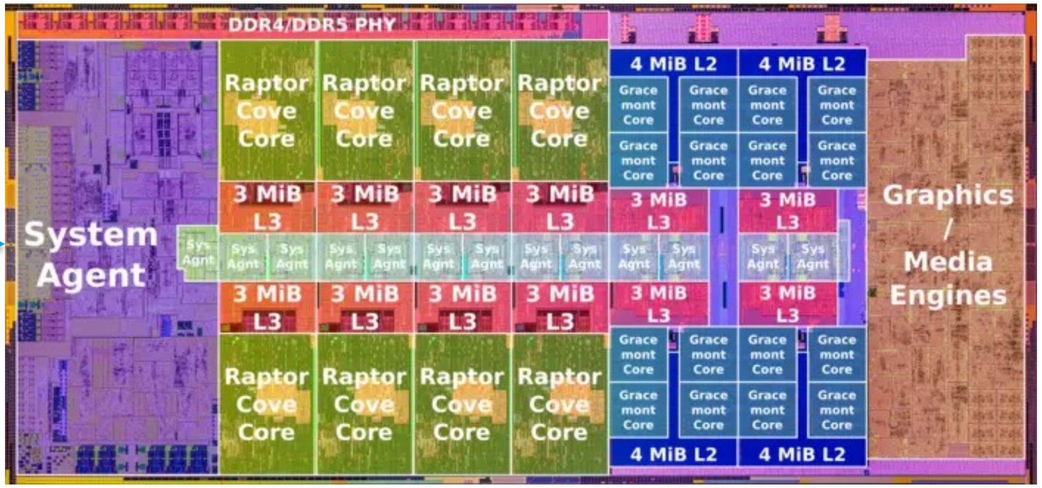


HOCHSCHULE ESSLINGEN

CACHES: IMPLEMENTATIONS

CPU-local Caches are a key-component to achieve performance:
The table lists generations of microprocessors (and it's cores/arch name)

Cache / Introduced?	Raptor Lake Core: Golden Cove 10/2022	Ice Lake Core: Sunny Cove 09/2019	Skylake Core: Broadwell 08/2015	AMD ZEN4 09/2022
L1 Data	48 kB, (12-way?)	48 kB, **12-way**	32 kB, 8-way	32 kB, 8-way
L1 Instruction	32 kB, (8-way?)	32 kB, 8-way	32 kB, 8-way	32 kB, 8-way
L2	**2 MB, 8-way (P)**	**512 kB, 8-way**	256 kB, 4-way	**1 MiB, 8-way**
L3/core	**3 MB, 16-way (P)**	2 MB, 16-way	2 MB, 16-way	**4 MB, 16-way**
µOp Cache	**4096**	2304	1536	**4096**



CACHES: SIMPLE MODEL OF HOW MUCH TO GAIN

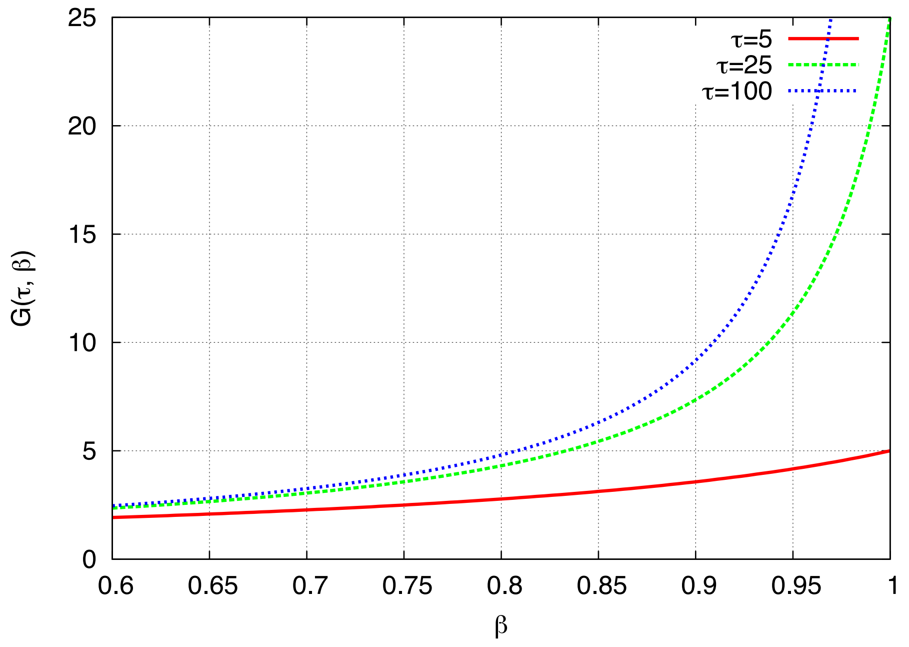
Assumption: Cache-Memory is τ faster than main memory.

- In this simplistic model, we do not account for latency or bandwidth, but just look at time: $T_m = \tau T_c$
- With a Cache-Hit/Miss ratio β, i.e. β is the ratio of accesses to already cached data, what gain is to be expected?
- The average time is:

$$T_{av} = \beta T_c + (1 - \beta) T_m$$

- The gain G for variable τ and β is therefore (shortened):

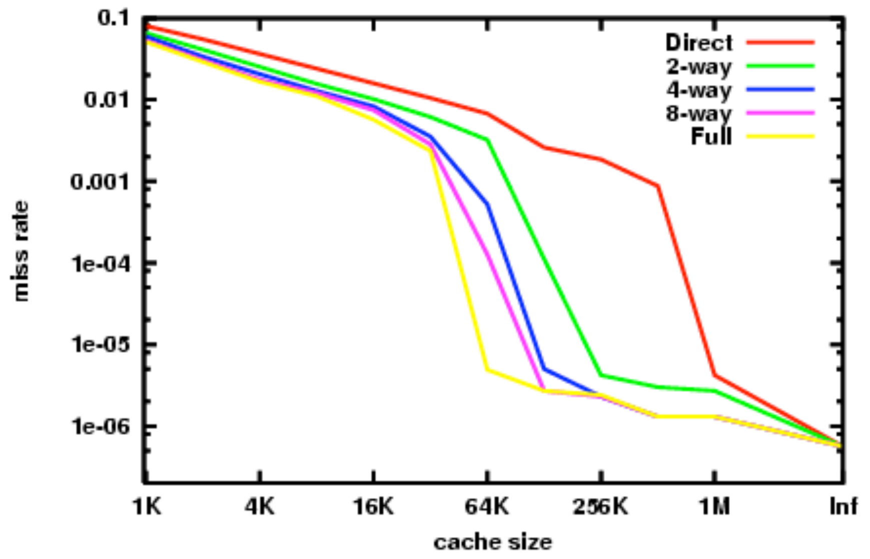
$$G(\tau, \beta) = \frac{T_m}{T_{av}} = \frac{\tau}{\beta + \tau(1 - \beta)}$$



CACHES: CURRENT IMPLEMENTATIONS…

Caches are tailored for specific uses:
- Instruction Cache
- Data Cache

Both have specific access patterns and typical working set sizes.
E.g. the working set size of Integer portion of SPEC2000 Benchmark:



Source: Wikipedia

- Current implementations have up to 4 levels of Cache…
- AMD Ryzen Zen 3 with 3D Stacking „V-Cache“:
  - 64x 32kB Level 1 Instruction Cache (8-way associative), write-back
  - 64x 32kB Level 1 Data Cache (8-way associative), write-back
  - 64x 512kB Level 2 combined Cache (8-way associative), write-back
  - 64 to **256 MB** Level 3 combined Cache (16-way associative, with hash)
- Discuss, how Instruction Cache access patterns differ from Data

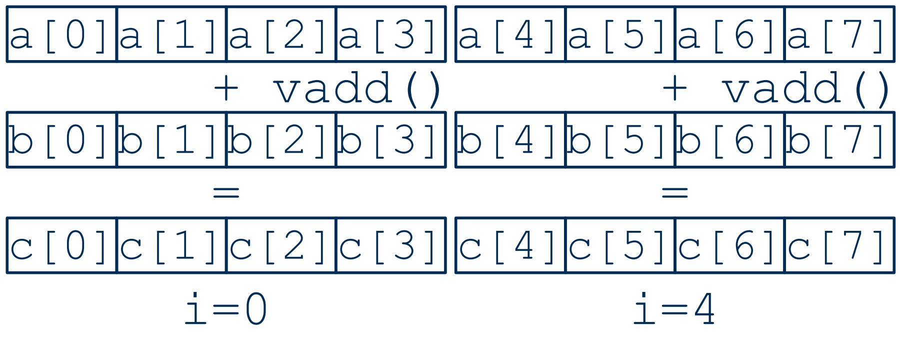
SIMD

SINGLE INSTRUCTION MULTIPLE DATA

Another big improvement in execution efficiency is SIMD:
1 instruction computes 2 (Intel SSE) or up to 64 (Intel AVX512) elements

This is an “old” concept: Vectorization (think NEC SX from 1990 ’til now), compilers & programmers nevertheless need to adapt their code!

```
for (int i = 0; i < N; i++) {
  c[i] = a[i] + b[i];
}
```



a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7]
+ vadd()
+ vadd()
b[0] b[1] b[2] b[3] b[4] b[5] b[6] b[7]
=
=
c[0] c[1] c[2] c[3] c[4] c[5] c[6] c[7]
i=0
i=4

Compiler needs to detect/allow:

- Loop unrolling (here by 4)
  (Is N divisible by 4? Handle N%4)
- Is N large enough?
- Are `a`, `b`, `c` properly aligned?
  Are they Pointers, and alias??

Result:

- ¼ of the (loop body’s) instructions (but not loop control).
- Better scheduling of core’s ressources

SIMD PROGRAMMING 1/2

HOCHSCHULE ESSLINGEN

Simple Loops, Simple data structures help!

1. Split Loops into multiple loops:

```
for (int i = 0; i < N; i++){
  if (a[i]%2)
   c[i] = a[i] + b[i];
  else
    c[i] = a[i] - b[i];
}
```

```
for (int i = 0; i < N; i++)
  if (a[i]%2 != 0)
    c[i] = a[i] + b[i];
for (int i = 0; i < N; i++)
  if (a[i]%2 == 0)
    c[i] = a[i] - b[i];
```

2. Use Struct of Arrays (SoA) instead of Array of Structs (AoS):

```
struct {
  int a;    // 4 Byte Padding
  double c;// 8 Byte Aligned
  float b;
} aos[1000];
```

```
struct {
  int a[1000];
  float b[1000];
  double c[1000];
} soa;
```

SIMD PROGRAMMING 2/2

Use the compiler to help & document the code:

Intel Compiler:

```
icc -vec-report=5 -c file.c     # Reports on vectorization of loops
icc -opt-report=5 -c file.c     # Reports optimization hints
```

GNU Compiler:

```
gcc -fverbose-asm -S file.c     # Generates verbose ASM output
gcc -fopt-info-vec-missed=vec.miss # Reports missed vectorization
```

Further options to get to vectorize:

1. Properly align types:

```
__declspec(align(64)) float a[1000];         // Windows
__attribute__((aligned(64))) float a[1000]; // Linux
```

   and `_mm_malloc()` / `_mm_free()` for dynamic allocation:

```
float * c = _mm_malloc(1000*sizeof(float), 64);
```

2. Use (compiler-specific) intrinsics, e.g.

```
#include <immintrin.h>
_mm512_add_epi64 (__m512i __A, __m512i __B);
```

SIMD RESOURCES & LINKS

The CPU vendors programming manual is a good starting point:

- Links for ARM’s NEON and SVE/SVE2
  SVE: Scalable Vector Extension:
  - 32 Vector regs z0 … z31, implementation defined: multiples of 128 Bit registers (with a max. of 2048 bits);
  - 16 predicate registers p0 … p15 to select vector lanes.

- Intel C++ Compiler Guide on Vectorization

- Intel Vector Intrinsics Guide

- To check different in compiler capabilities, use the Compiler Explorer

AMD APUs
Am Beispiel AMD MI300A

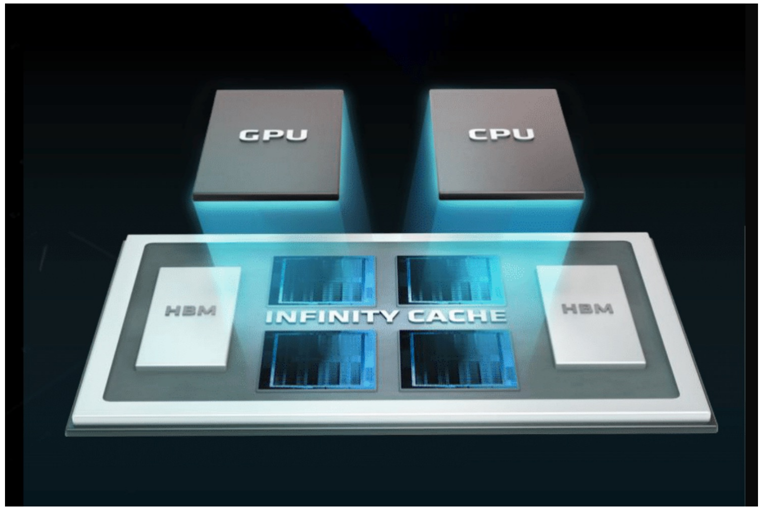
AMD APU

AMD Accelerated Processing Unit (APU) combine CPU & iGPU **since 2012** (e.g. used in PlayStation 4 & 5, Steam Deck)

- The main advantage is shared addressing space
- Due to the different design (ISA, memory access patterns and memory synchronization) it's a Heterogeneous System Architecture (HSA)

(news June / November 2023):
@SC2023: AMD Instinct MI300A (APUs) power LLNL's HPC "EL Capitan":

- **2 ExaFlops**, i.e. $2*10^{18}$ FP32 OP/s
- Due to massive Power of 24 core Zen 4 &
- GPU: CDNA-3 &
- Massive Memory Bandwidth (to 128 GB HBM3 RAM on package): **5,218 TB/s!!!**



- **According to AMD ML benchmarks 8x faster than MI250 GPUs**

HOCHSCHULE ESSLINGEN

LECTURE: LITERATURE

According to list of literature:

- Patterson, D., Hennessy, J.: *Computer Organization and Design*, Kaufmann, 2011
  (Deutsche Übersetzung: Rechnerorganisation und –entwurf, Spektrum)
- Hennessy, J., Patterson, D.: *Computer Architecture: A quantitative Approach, **5th edition**,* Morgan Kaufmann, 2017
- Tanenbaum, A., Austin, T.: *Structured Computer Organization*, Pearson, 6th edition, 2013
- Tanenbaum, A., Austin, T.: *Rechnerarchitektur: von der digitalen Logik zum Parallelrechner*, Pearson, 6th ed., 2014
- Huang, H.W.: *The HCS12/9S12. An Introduction to the HW and SW interface*, Thomson Learning, 2009
- Beierlein, T.: *Taschenbuch Mikroprozessortechnik*, Carl-Hanser Verl., 2011