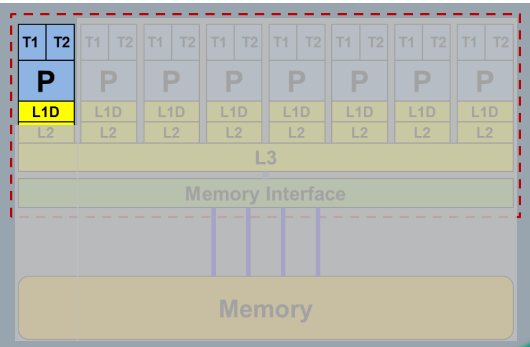
Programming Techniques for Supercomputers: Modern processors: Single Core



Introduction

- Basic technology trend / Moore‘s law
- Basic concept of single core architecture

Key single core features

- Pipelining
- Superscalarity
- SingleInstructionMultipleData

Maximum In-Core Performance

Prof. Dr. G. Wellein[a,b] , Dr. G. Hager[a]

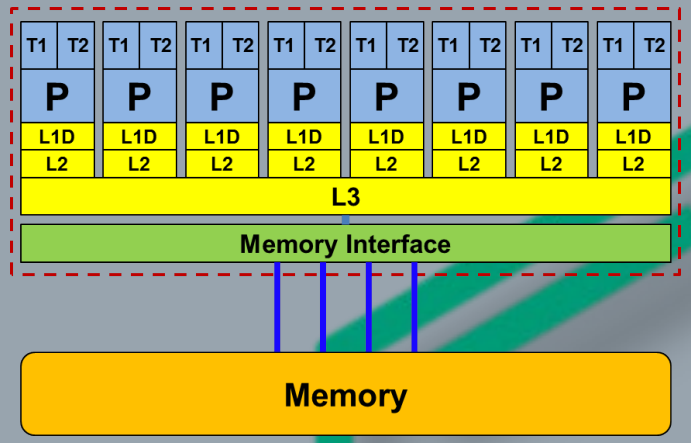
[a]Erlangen National Center for High Performance Computing
[b]Department für Informatik

Friedrich-Alexander-Universität Erlangen-Nürnberg, Sommersemester 2025

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Programming Techniques for Supercomputers
Modern processors: Single Core



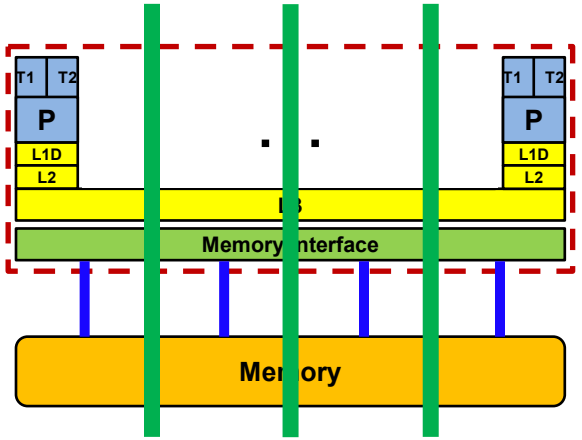
- Introduction
  - Basic technology trend / Moore's law
  - Basic concept of core architecture
- Key single core features:
  - Pipelining
  - Superscalarity
  - SingleInstructionMultipleData
- Maximum In-Core Performance

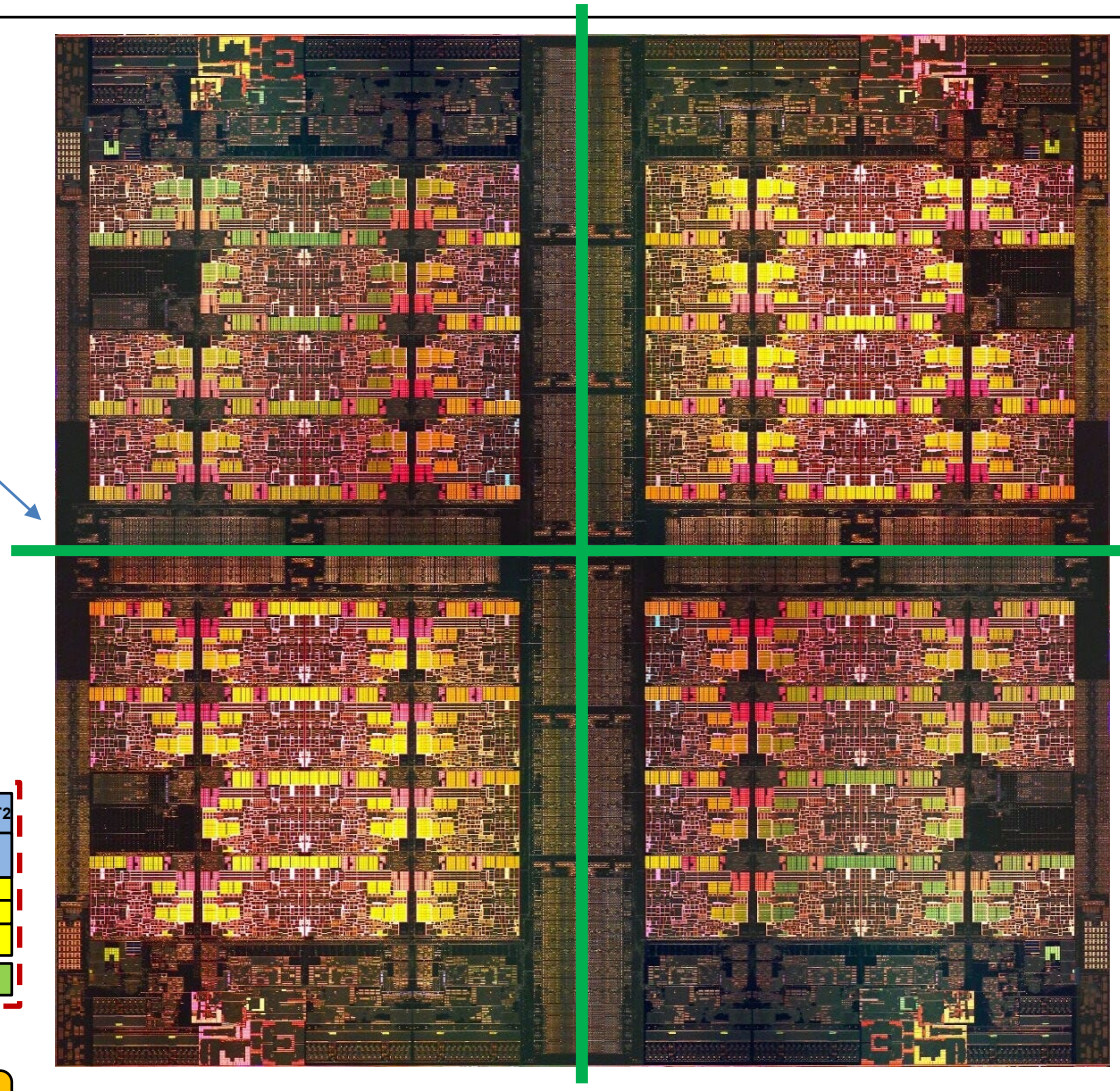
Multi-core today: Intel Xeon Sapphire Rapids (2023)

- Xeon “Sapphire Rapids” (Platinum/Gold/Silver/Bronze):
  Up to 60 cores running at 1.7+ GHz
  (+ “Turbo Mode” 4.8 GHz),
- Simultaneous Multithreading
  → reports as 120-way chip
- “Intel 7” process / up to 350 W
- Multi-die package (4 chips)
- Clock frequency:
  flexible ☺

Optional: “Sub-NUMA Clustering” (SNC) mode boot option

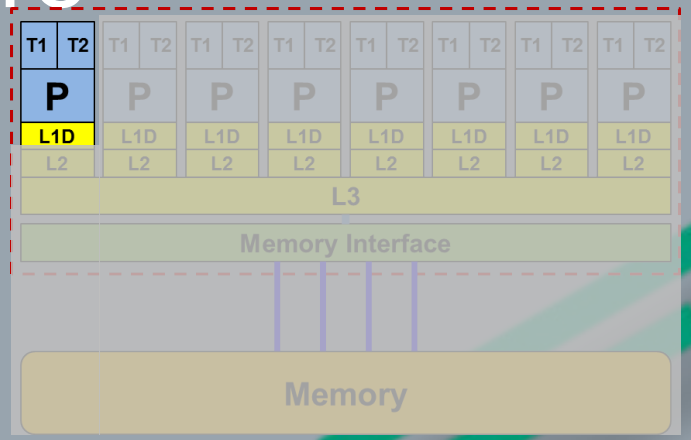
→ One memory domain per die





https://www.techpowerup.com/292204/intel-sapphire-rapids-xeon-4-tile-mcm-annotated

Programming Techniques for Supercomputers
Modern processors: Single Core



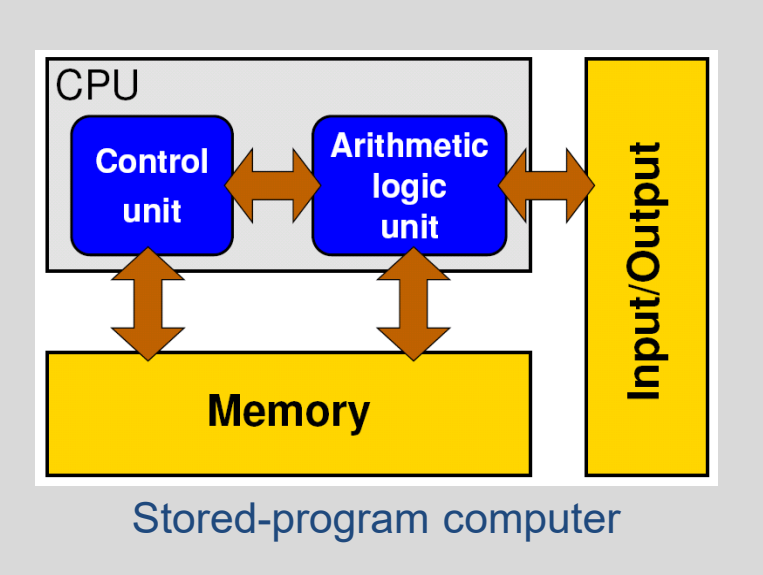
Introduction
- Basic technology trend / Moore's law
- Basic concept of core architecture

Key single core features:
- Pipelining
- Superscalarity
- SingleInstructionMultipleData

Maximum In-Core Performance

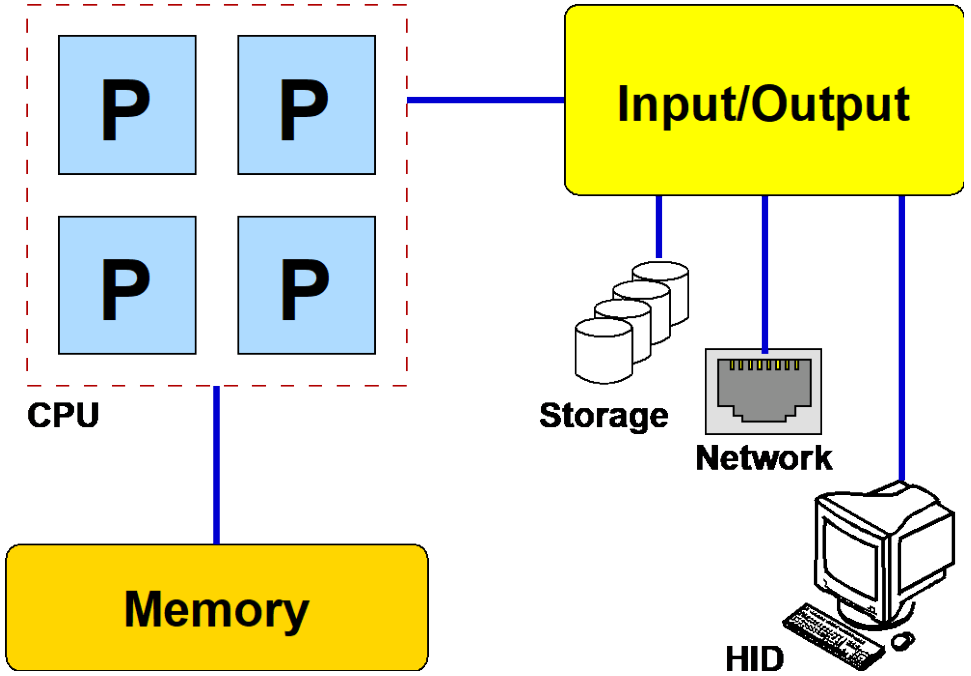
Basic „stored program computer“ concept – still in use

- Stored Program Computer” concept (Turing 1936)



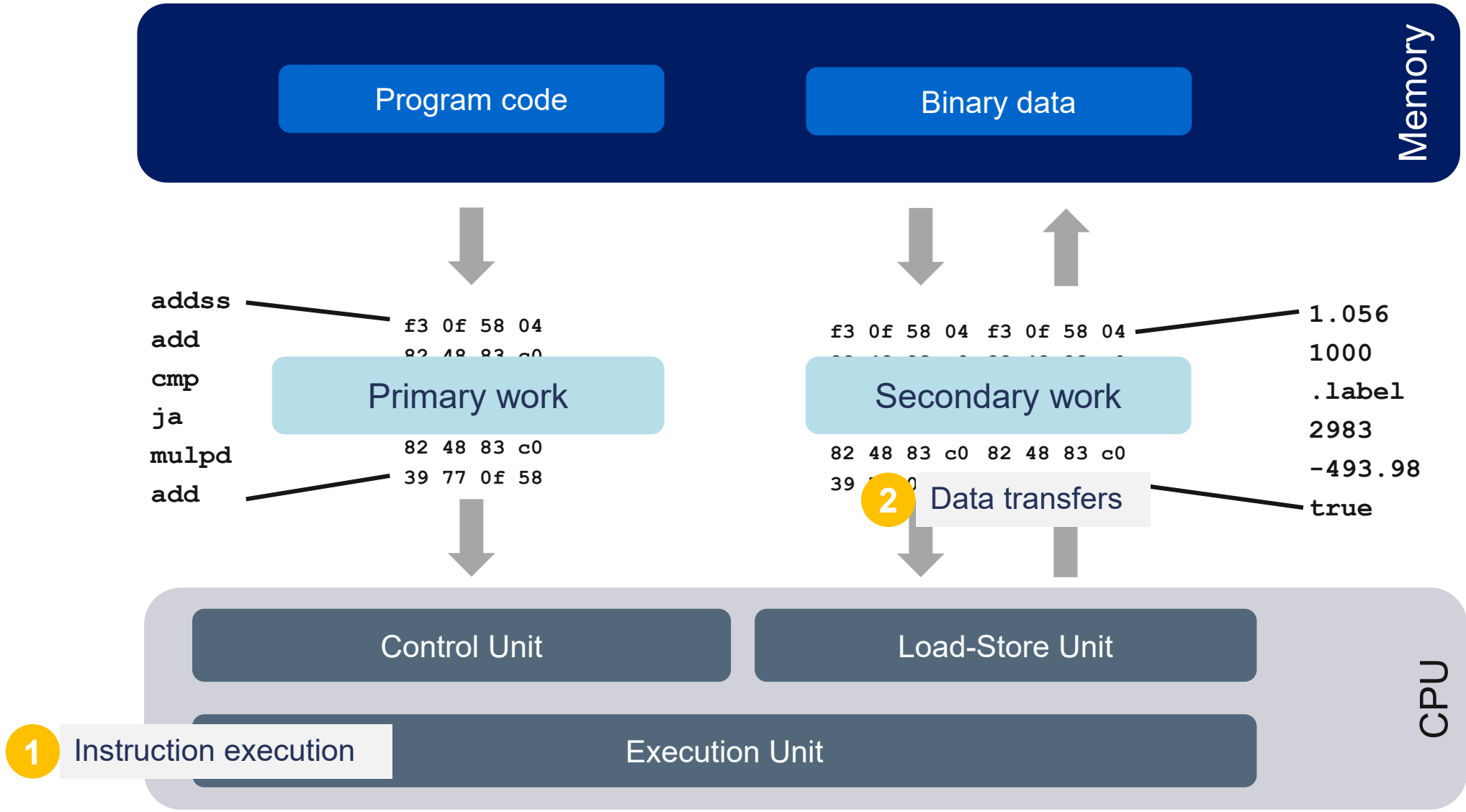
Stored-program computer

- Similar designs on all modern systems

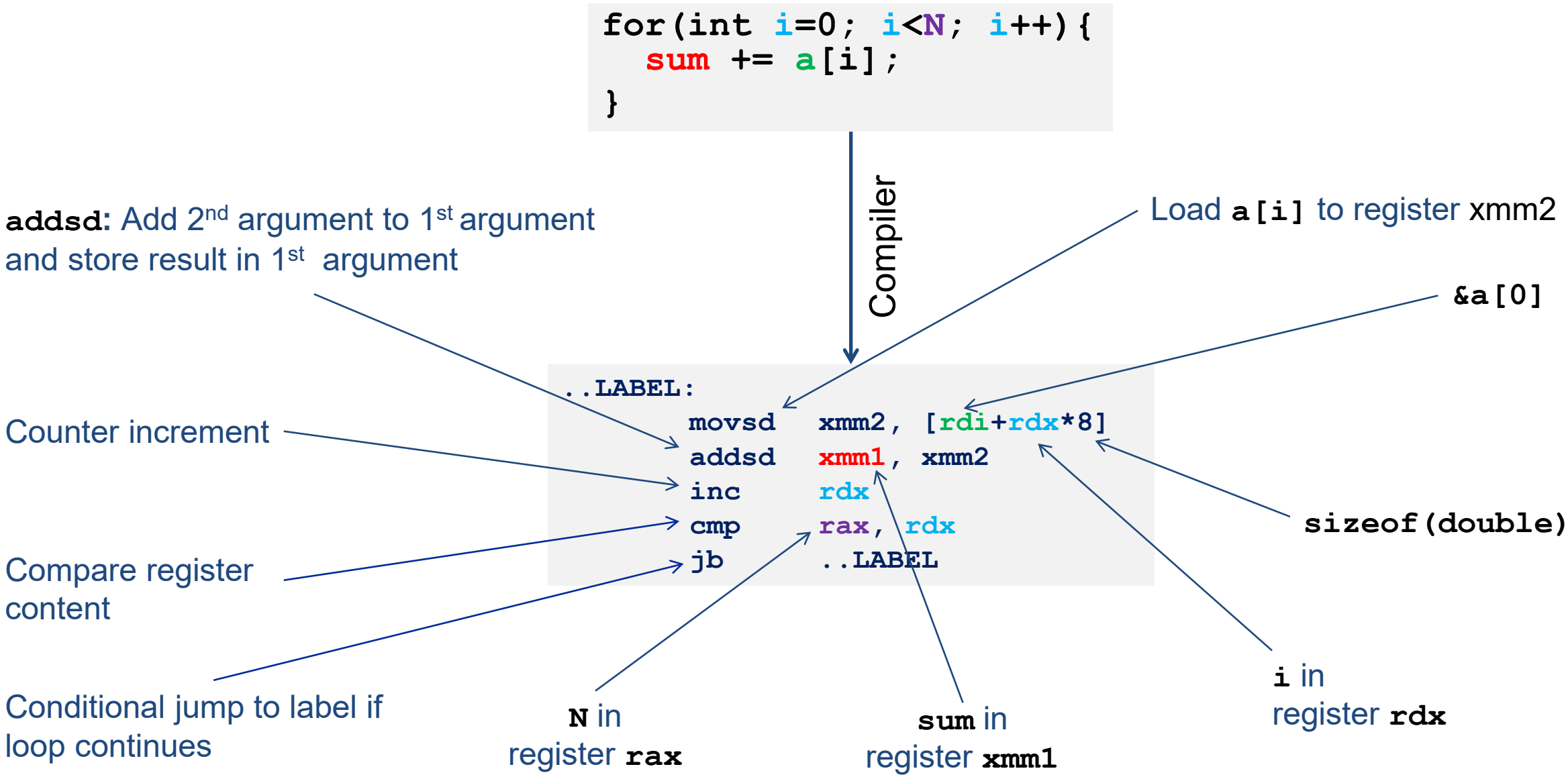


Flexibility!
(Still) multiple potential bottlenecks

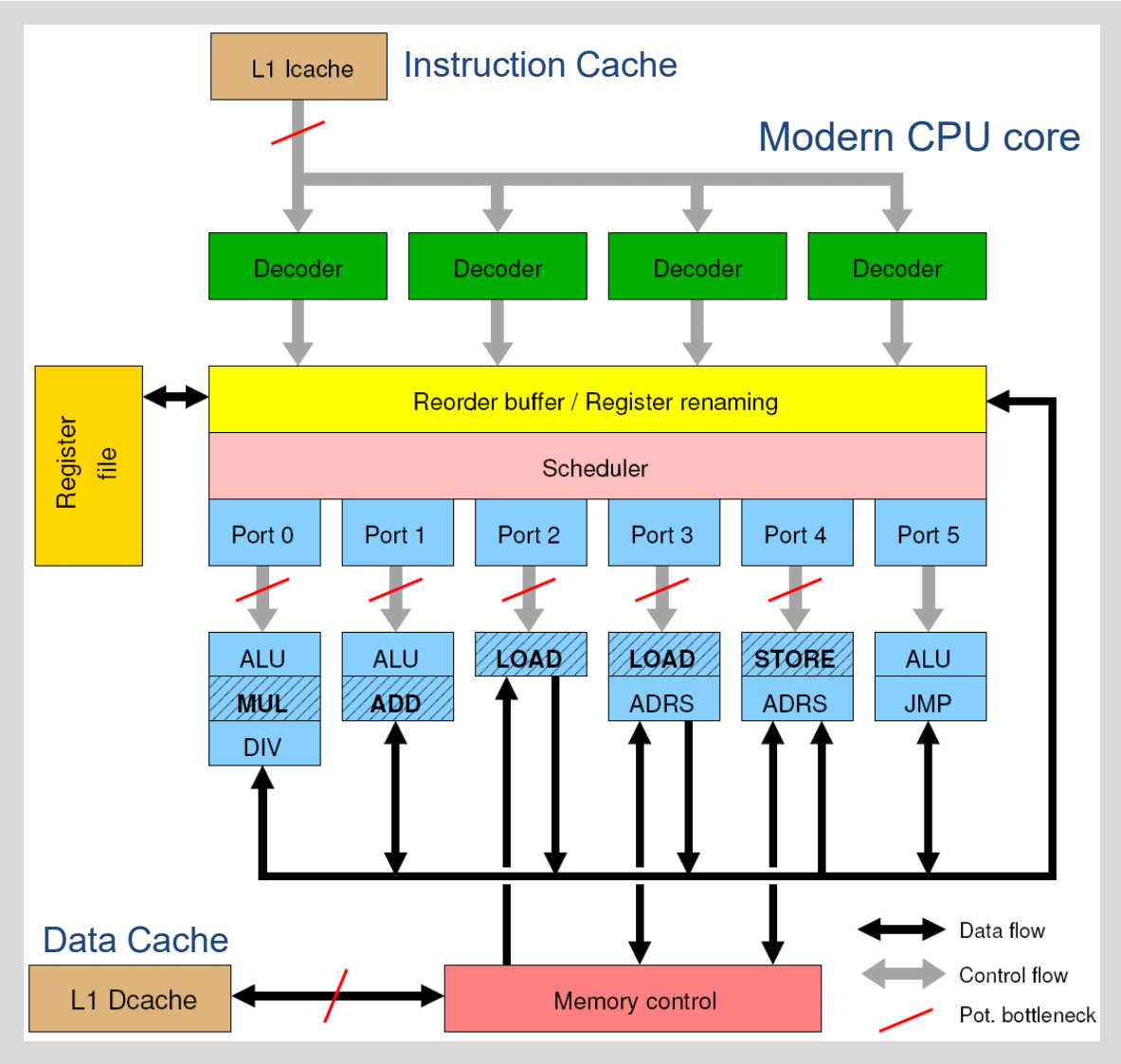
Stored Program Computer

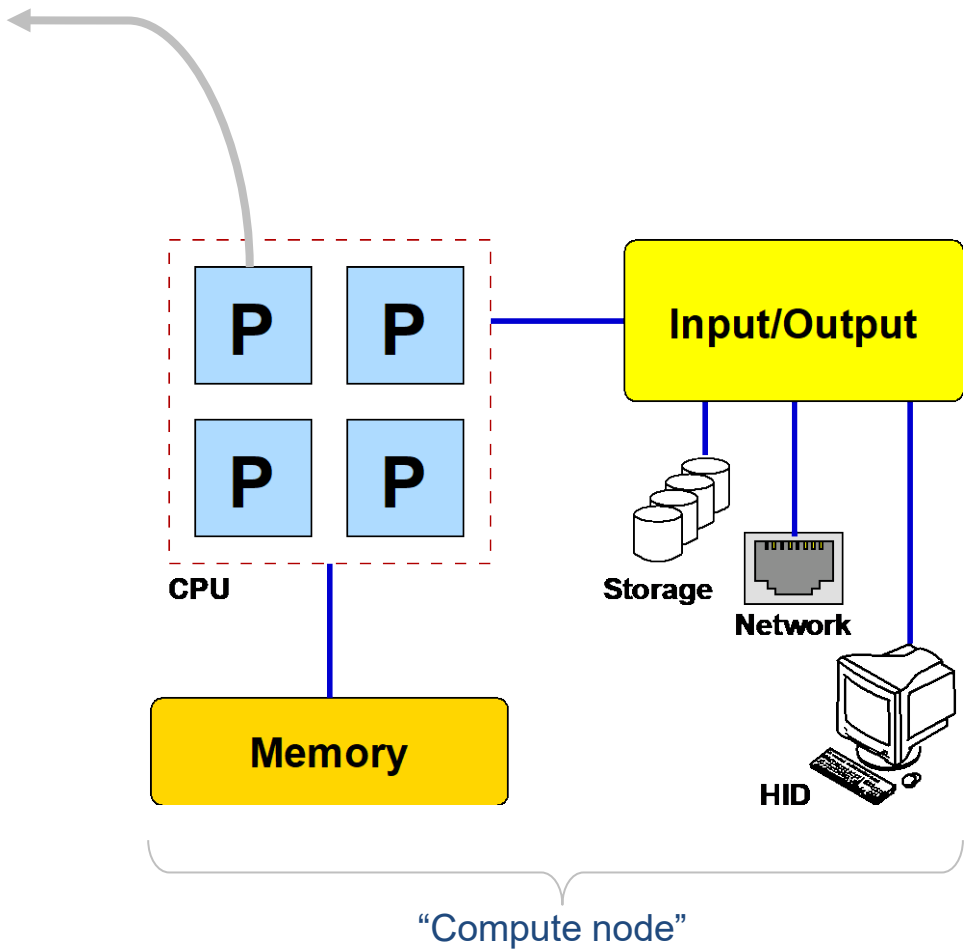


From high level code to actual execution

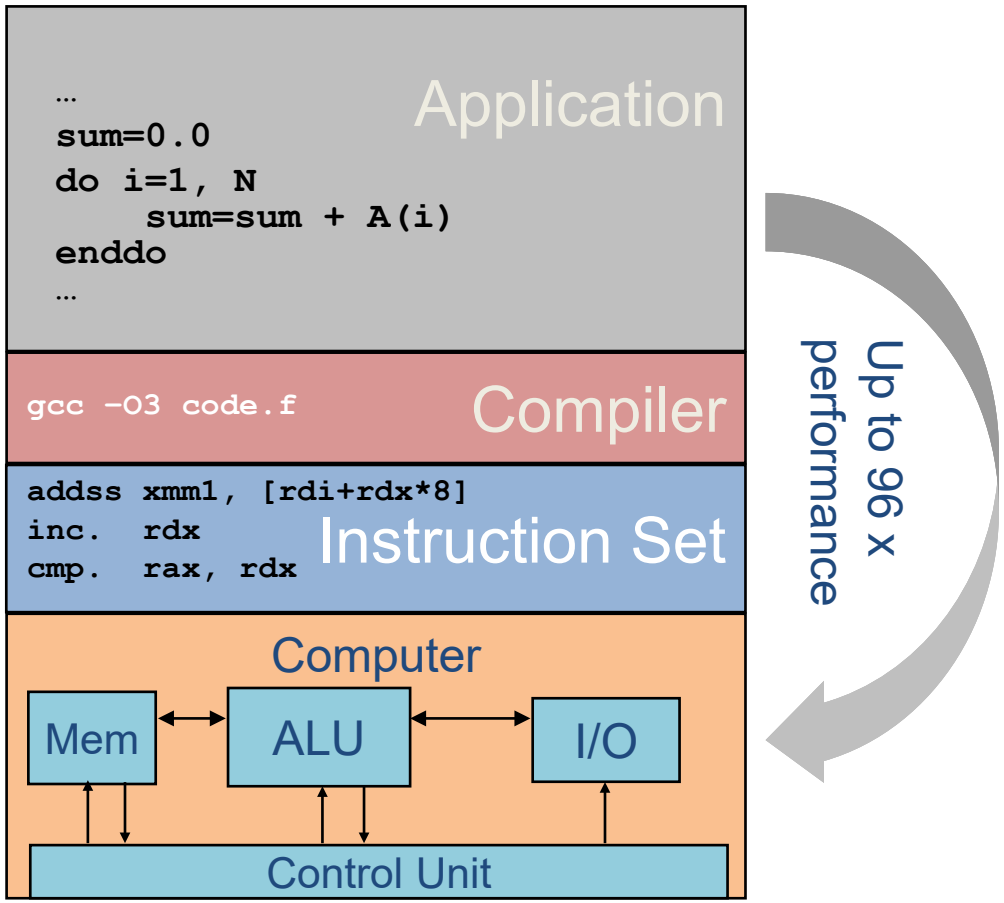


General-purpose (cache based) microprocessor core





Introduction: From application to microprocessor core



- High Level Programming Language (e.g. C / C++ / Fortran): Aplication – portable
- Compiler translates program to Instruction set (architecture) (IA32, Intel 64, AMD64 a.k.a. x86, x86_64)
- Instruction Set Architecture (ISA): Hardware specific

Introduction: Instruction Set Paradigms

- In the beginning (60's): Complex Instruction Set Computers (CISC) :
  - Powerful & complex instructions
  - Instruction set is close to high-level programming language
  - Variable length of instructions - Save storage!

```
MULT r0 * [a2] → [a1]
```

Multiply content of address `a2` with
register content `r0`
and write back to address `a1`

- Mid 80´s: Reduced Instruction Set Computer (RISC) evolved:
  - Fixed instruction length; enables pipelining and high clock frequencies
  - Uses simple instructions, e.g., above instruction is split into at least 3 instructions:

```
LOAD [a2]→r1; MULT r0*r1→r2; STORE r2→[a1]
```

- Nowadays: RISC processor cores
  - Almost…

x86 CISC/RISC hybrid

- Current x86_64 processors (Intel, AMD): Compiler still generates CISC instructions; but processor core is RISC-like
  - Example:

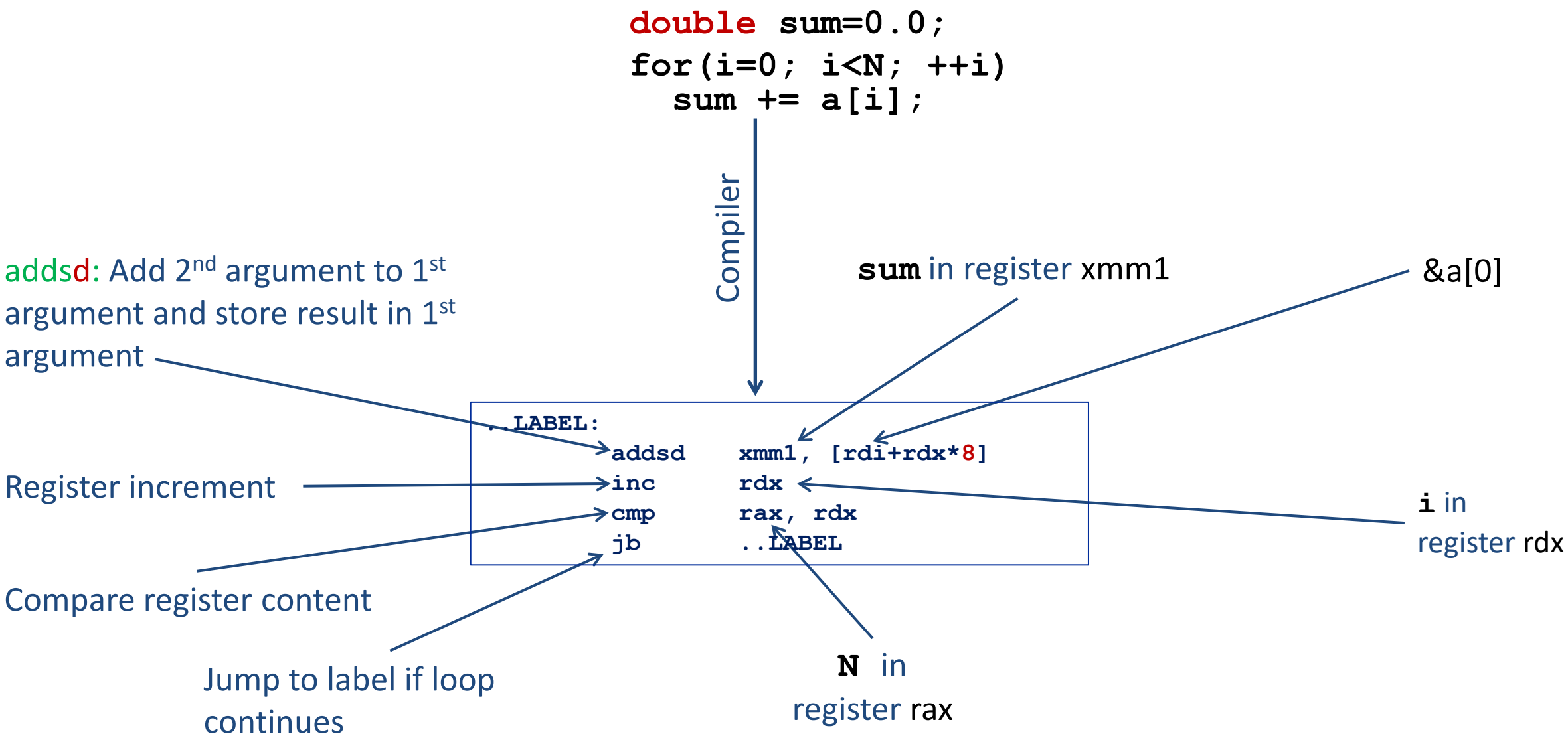
```
addsd     xmm1, [rsi+rax*8]
```

`xmm1:` register holding floating point data
`rsi, rax:` register holding integer data

→ combined address calculation, LD, and ADD instruction

1. Calculate address `rsi+rax*8`
2. Load double value from that address
3. Add double value into register `xmm1` (accumulate)

From high level code to machine execution (CISC-style)



From high level code to macro-/microcode execution

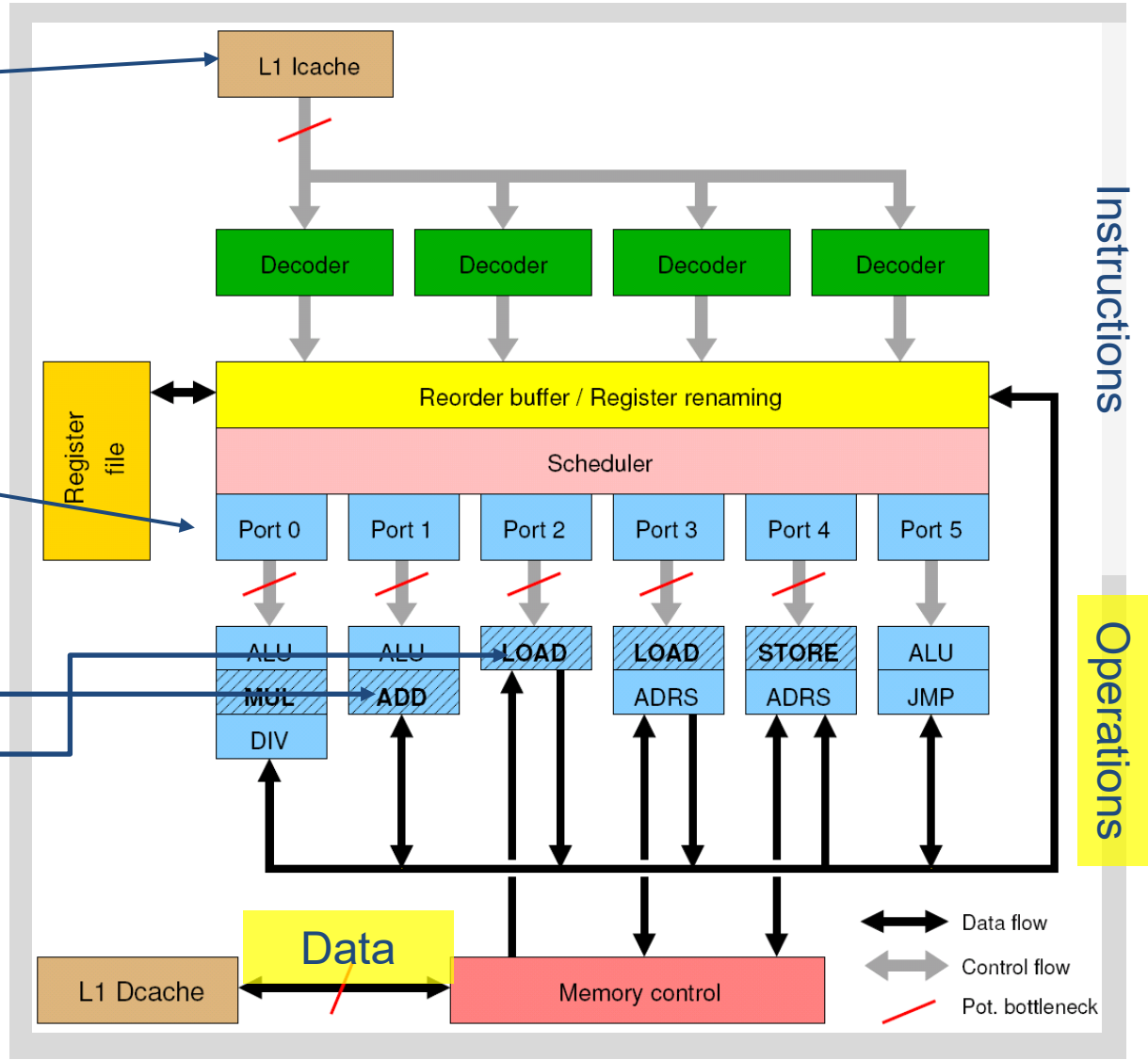
```
..LABEL:
   addsd    xmm1, [rdi+rdx*8]
   inc      rdx
   cmp      rax, rdx
   jb       ..LABEL
```

Instructions are mapped to execution ports / units

ADDSD Instruction requires

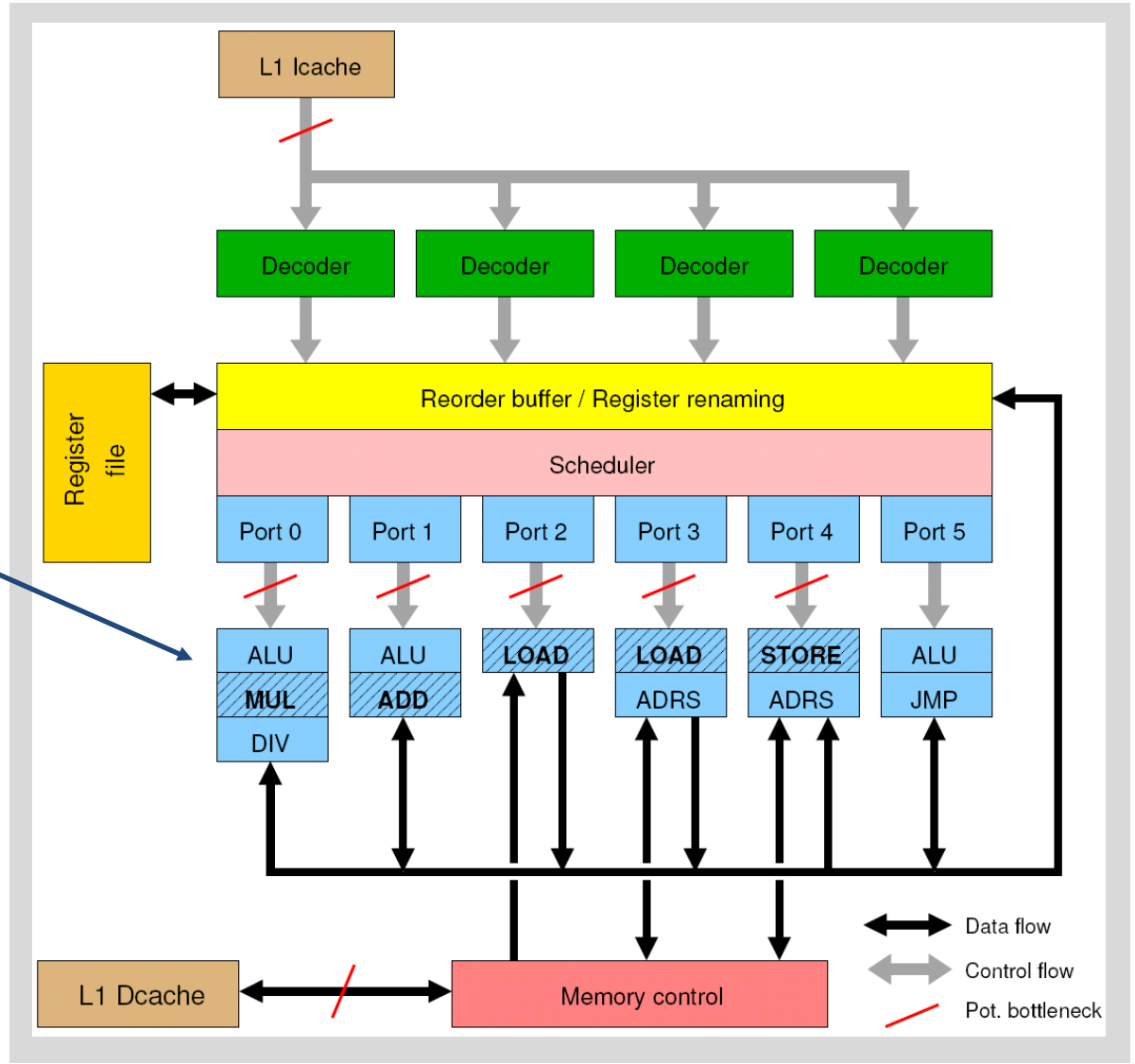
LOAD Execution unit

ADD Execution unit



Key single-core features: Pipelining

Pipelining
Most units can complete
one instruction per cycle, e.g.
MULT / ADD / LOAD /STORE



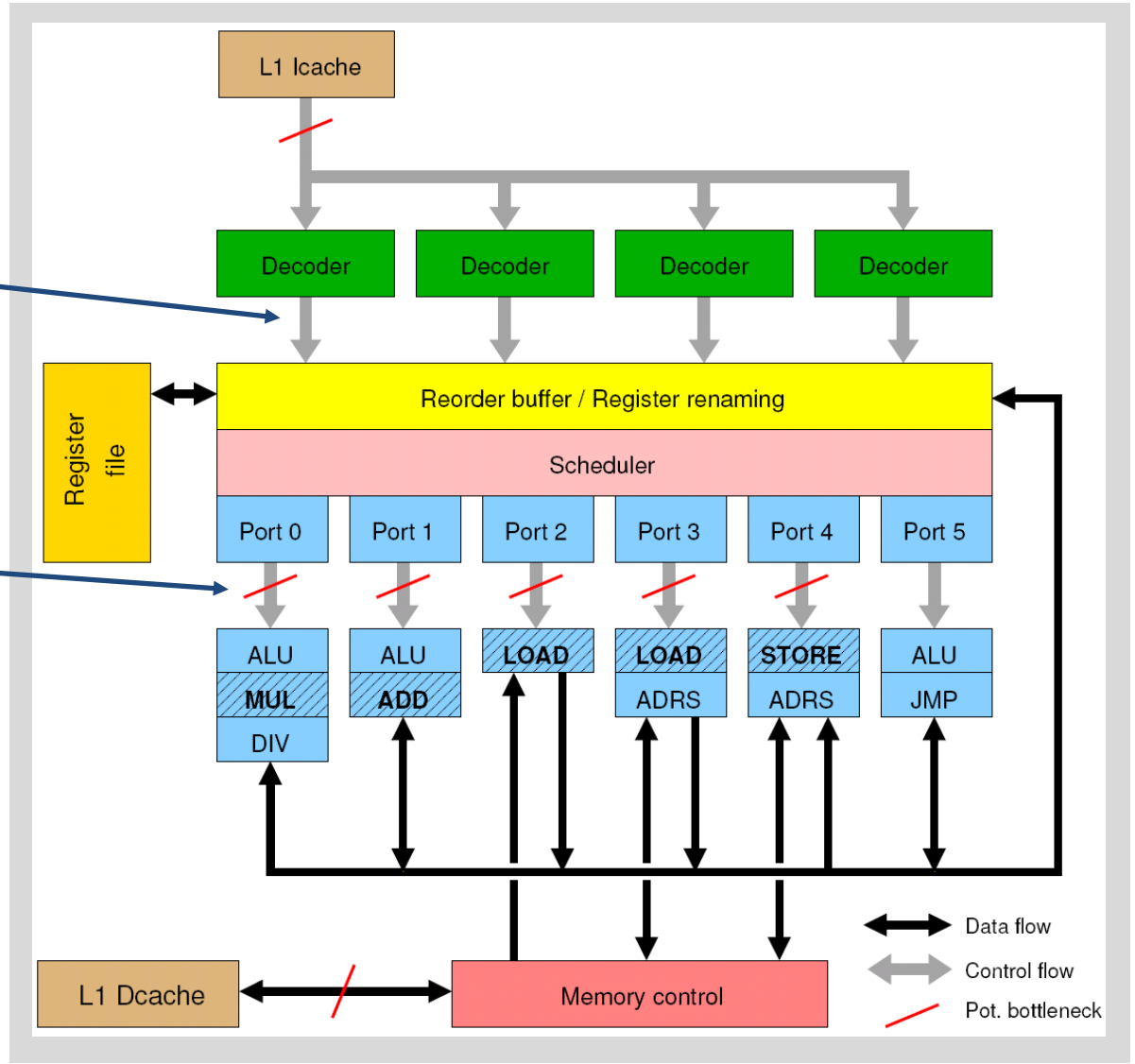
Focus on: Floating Point
Instructions/Operations

Key single-core features: Superscalarity

Multiple instructions issued in parallel

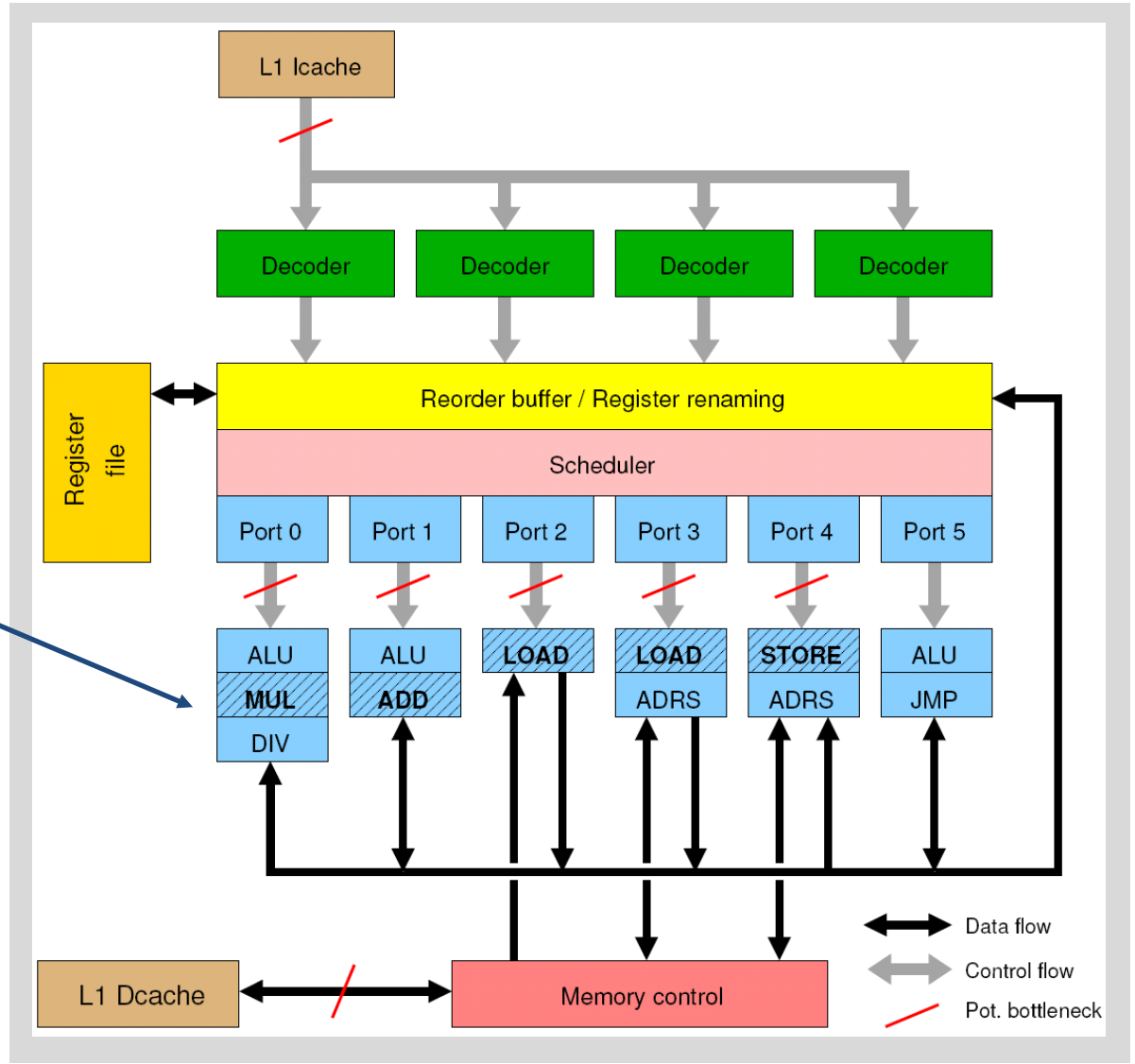
Multiple execution units/ports (can run in parallel)

Superscalarity
(“Instruction level parallelism“)

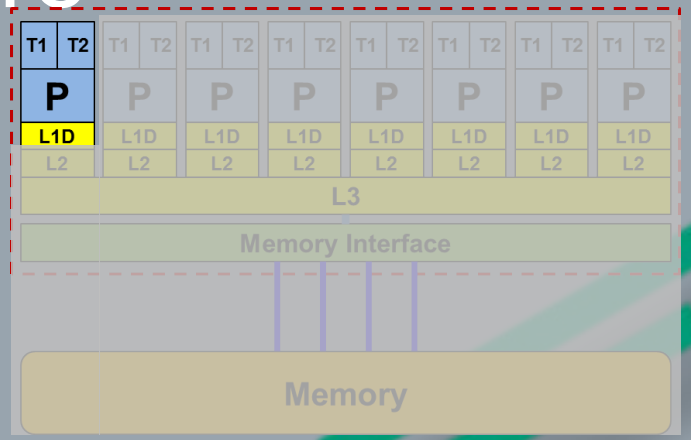


Key single-core features: SIMD

SIMD:
Single Instruction Multiple Data
Instruction is applied to multiple operands in parallel („width of execution units/registers“)



Programming Techniques for Supercomputers
Modern processors: Single Core



- Introduction
  - Basic technology trend / Moore's law
  - Basic concept of core architecture
- Key single core features:
  - Pipelining
  - Superscalarity
  - SingleInstructionMultipleData
- Maximum In-Core Performance

Pipelining of arithmetic/functional units

- Concept:
  - Split complex instruction into several simple / fast steps (stages)
  - Each step takes the same amount of time, e.g. a single cycle
  - Execute different steps on different instructions at the same time (in parallel)

- Benefit:
  - Pipeline can work on multiple instructions simultaneously (in parallel)
  - If pipeline is full one instruction completes every cycle → Throughput: 1 inst./cy.
  - Enables faster clock speeds (simple steps/stages)

- Drawback:
  - Pipeline must be filled ("wind-up") → start-up "latency" = number of stages
  - Independent instructions required → complex instruction scheduling by hardware ("out-of-order") or compiler ("software-pipelining")

- Pipelining is widely used in modern computer architectures
- Pipelining addresses Instruction Level Parallelism

Interlude: Possible stages for Floating Point Multiply

- Real numbers can be represented as mantissa and exponent in a “normalized” representation, e.g.: $s*0.m * 10^e$ with

  Sign $s=\{-1,1\}$

  Mantissa m which does not contain 0 in leading digit

  Exponent e some positive or negative integer

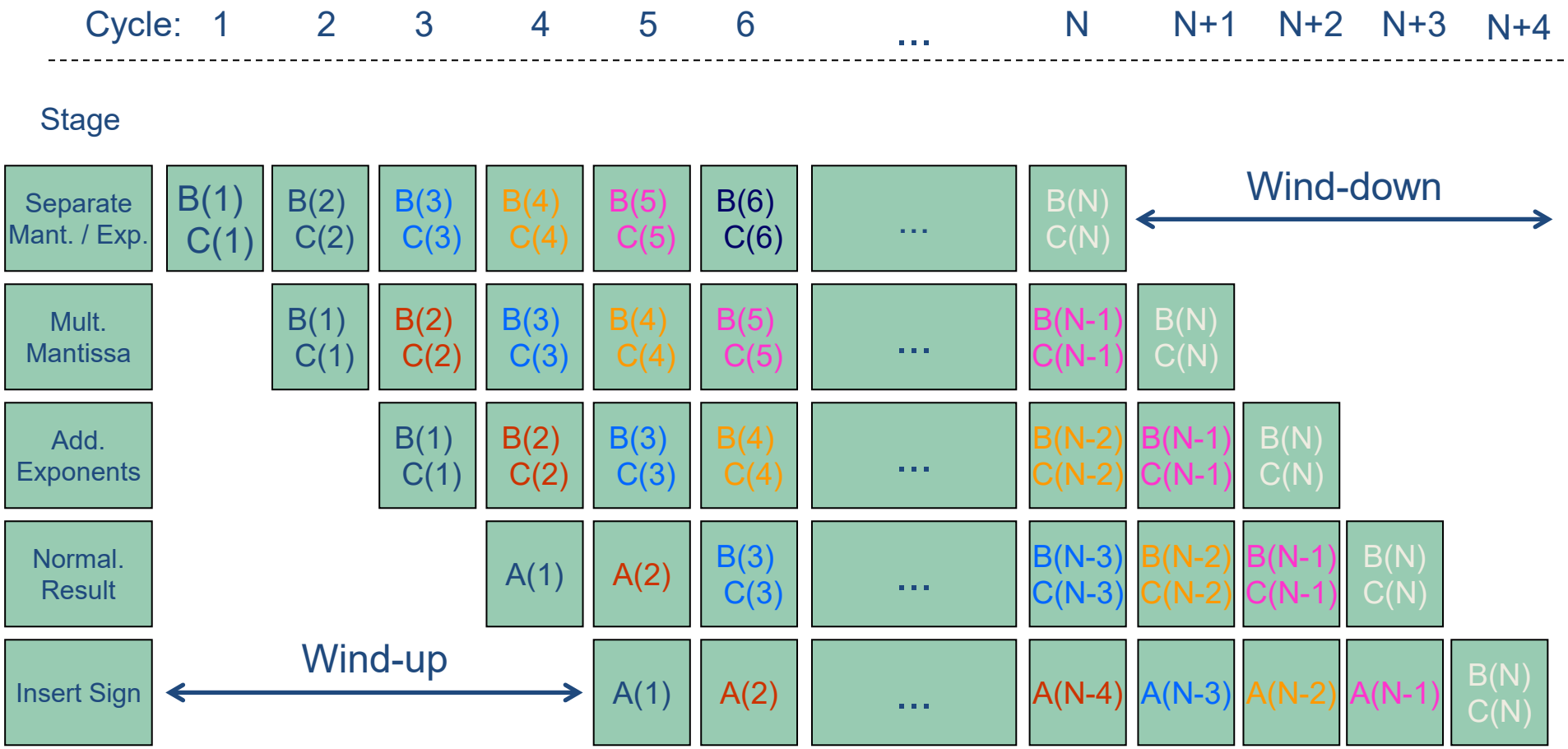
- Multiply two real numbers $r1*r2 = r3$
  $r1=s1*0.m1 * 10^{e1}$, $r2=s2*0.m2 * 10^{e2}$:

  $s1*0.m1 * 10^{e1} * s2*0.m2 * 10^{e2}$

→ $(s1*s2)* (0.m1*0.m2) * 10^{(e1+e2)}$

→ Normalize result: $s3* 0.m3 * 10^{e3}$

5-stage Multiplication-Pipeline: A(i)=B(i)*C(i) ; i=1,...,N



First result is available after 5 cycles (=latency of pipeline)!

After that one instruction is completed in each cycle (N-1 cycles)!

Empty pipeline stages in Wind-up/-down phase!

Pipelining: Latency, Throughput and Speed-Up

- Assume m-stage pipeline (**pipeline latency**: m cycles), fixed clock speed and N independent instructions to be executed

- Speed-up of pipelined (T_{pipe}) vs. non-pipelined (T_{seq}) execution time

$$\frac{T_{seq}}{T_{pipe}} = \frac{m \cdot N}{m + N - 1}$$

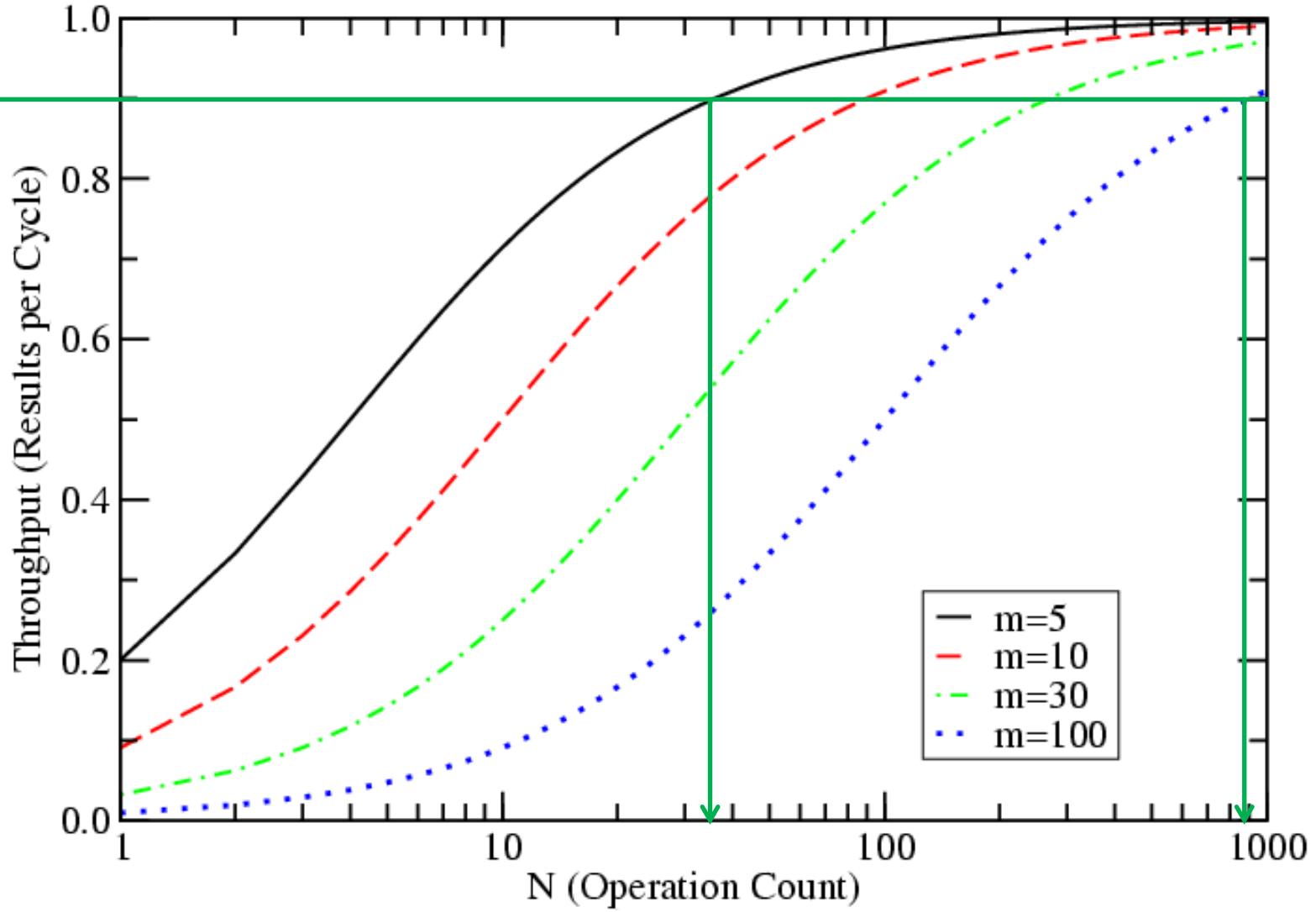
- **Pipeline throughput**, i.e. average instructions completed per cycle [inst./cy]:

$$\frac{N}{T_{pipe}} = \frac{N}{N + m - 1}$$

- Large N limits:
  - Speed-Up: ${}^{T_{seq}}/_{T_{pipe}} \approx m$ for $N \gg m$
  - Throughput: ${}^{N}/_{T_{pipe}} \approx 1 \frac{inst.}{cy}$ for $N \gg m$

Throughput as function of pipeline stages

90% pipeline efficiency



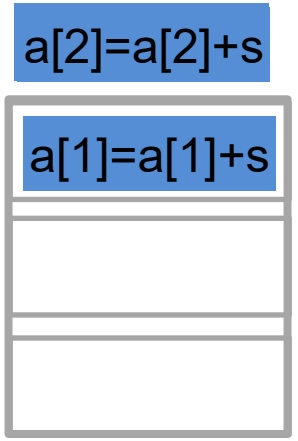
m = #pipeline stages

Efficient use of Pipelining

- (Potential) dependencies within loop body may prevent efficient software pipelining or OOO execution, e.g.:

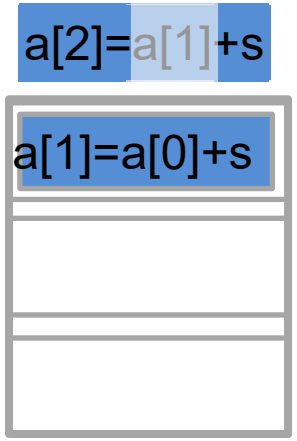
No dependency:

```
do i=1,N
   a(i) = a(i) + s
end do
```



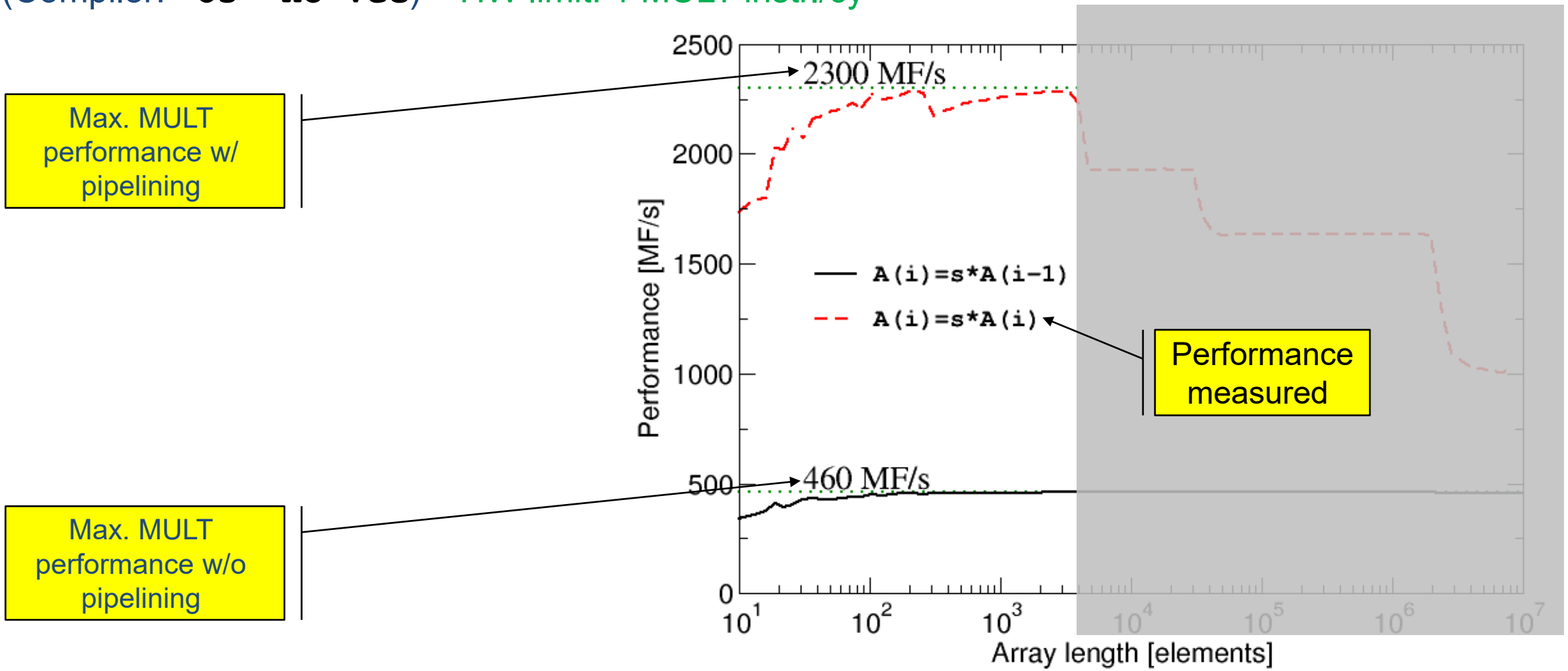
Dependency:

```
do i=2,N
   a(i) = a(i-1) + s
end do
```



Pipelining: Data dependencies

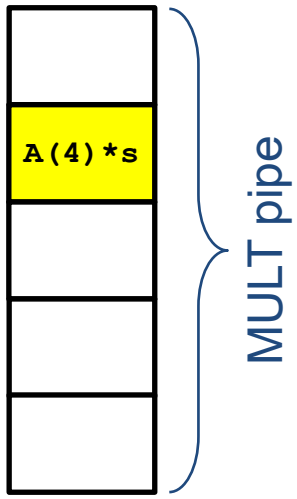
- Single core on Intel Xeon E5-2695 v3 (“Haswell”) with clock speed fixed to 2.3 GHz (Compiler: `-O3 -no-vec`) – HW limit: 1 MULT instr./cy



Pipelining: Data dependencies – performance model

Dependency

```
do i=2,N
   A(i) = A(i-1) * s
end do
```



Latency (MULT): 5 cy

Throughput: 1 MULT/5 cy

Clock Speed: 2.3 Gcy/s

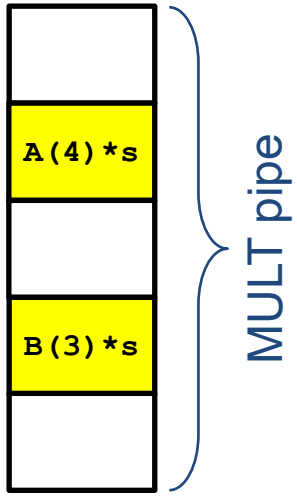
Performance:
$2.3 \cdot 10^9$ cy/s · 1 MULT / 5 cy

= 2300/5 MF/s = 460 MF/s
(with 1 MULT = 1 F)

Pipeline utilization / performance improvement by unrelated workload:

2 Dependencies

```
do i=2,N
   A(i) = A(i-1) * s
   B(i) = B(i-1) * s
end do
```



Latency (MULT): 5 cy

Throughput: 2 MULT/5 cy

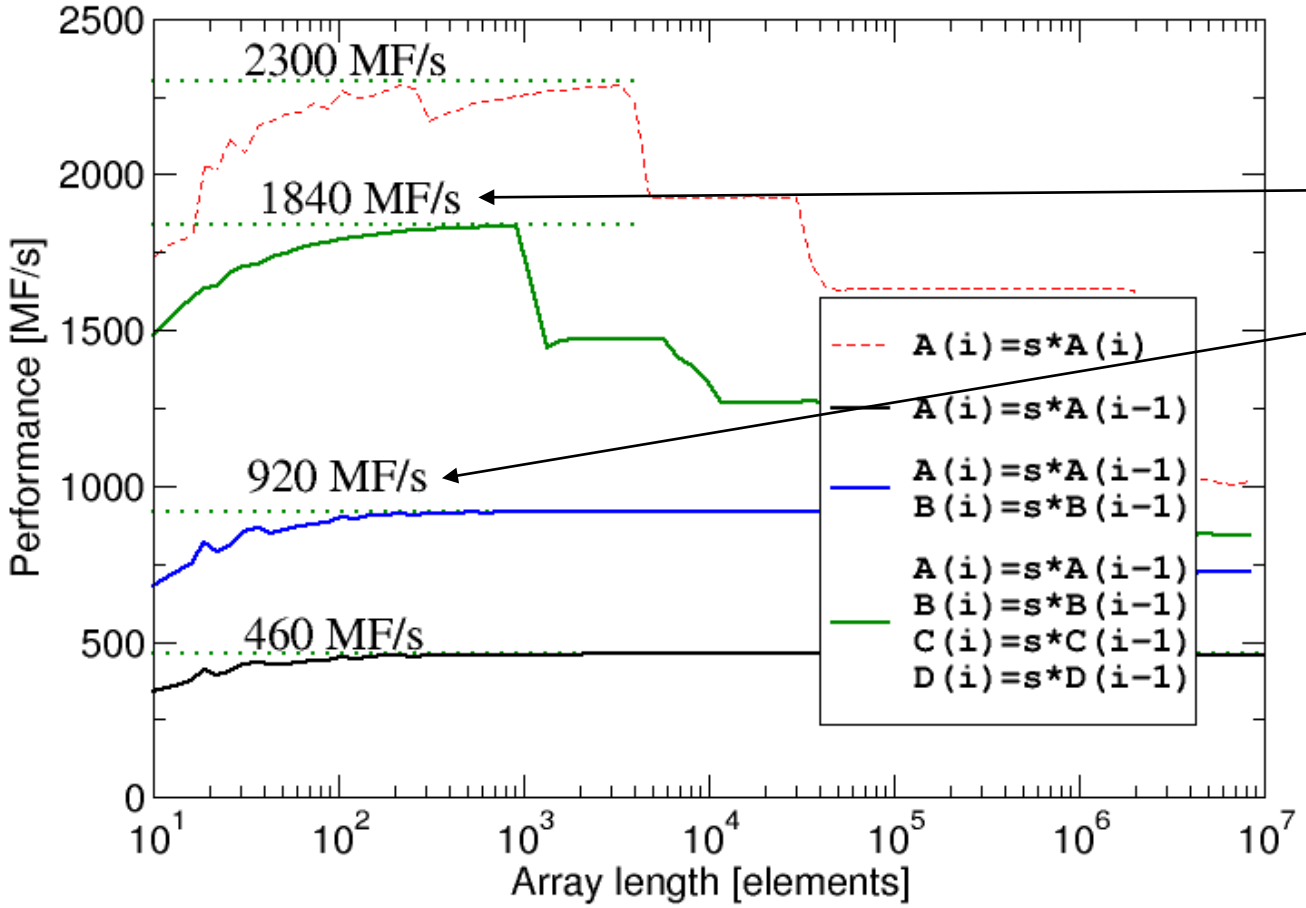
Clock Speed: 2.3 Gcy/s

Performance:
$2.3 \cdot 10^9$ cy/s · 2 MULT / 5 cy

= 2300 · 2/5 MF/s = 920 MF/s
(with 1 MULT = 1 F)

Pipelining: Data dependencies

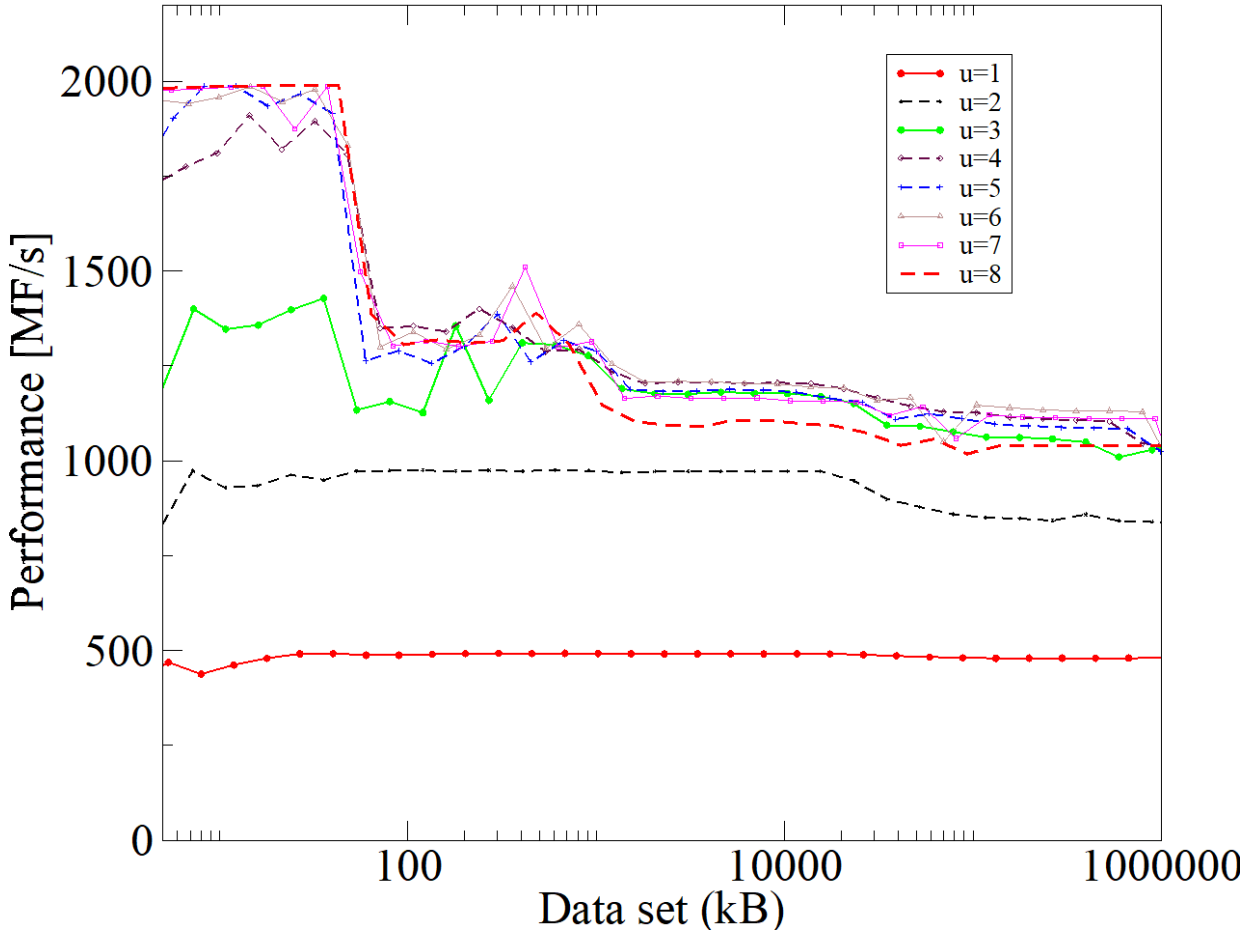
- Single core on Intel Xeon E5-2695 v3 (“Haswell”) with clock speed fixed to 2.3 GHz (Compiler: `-O3 -no-vec`) – HW limit: 1 MULT instr./cy



Increasing number of
„independent dependencies“
(i.e. increasing parallel workload)
improves pipeline throughput

Pipelining: Data dependencies

- Single core on Intel(R) Xeon(R) Platinum 8360Y CPU with clock speed fixed to **2.0 GHz** (Compiler: `-O3 -no-vec`) – HW limit: 2 MULT instr./cy



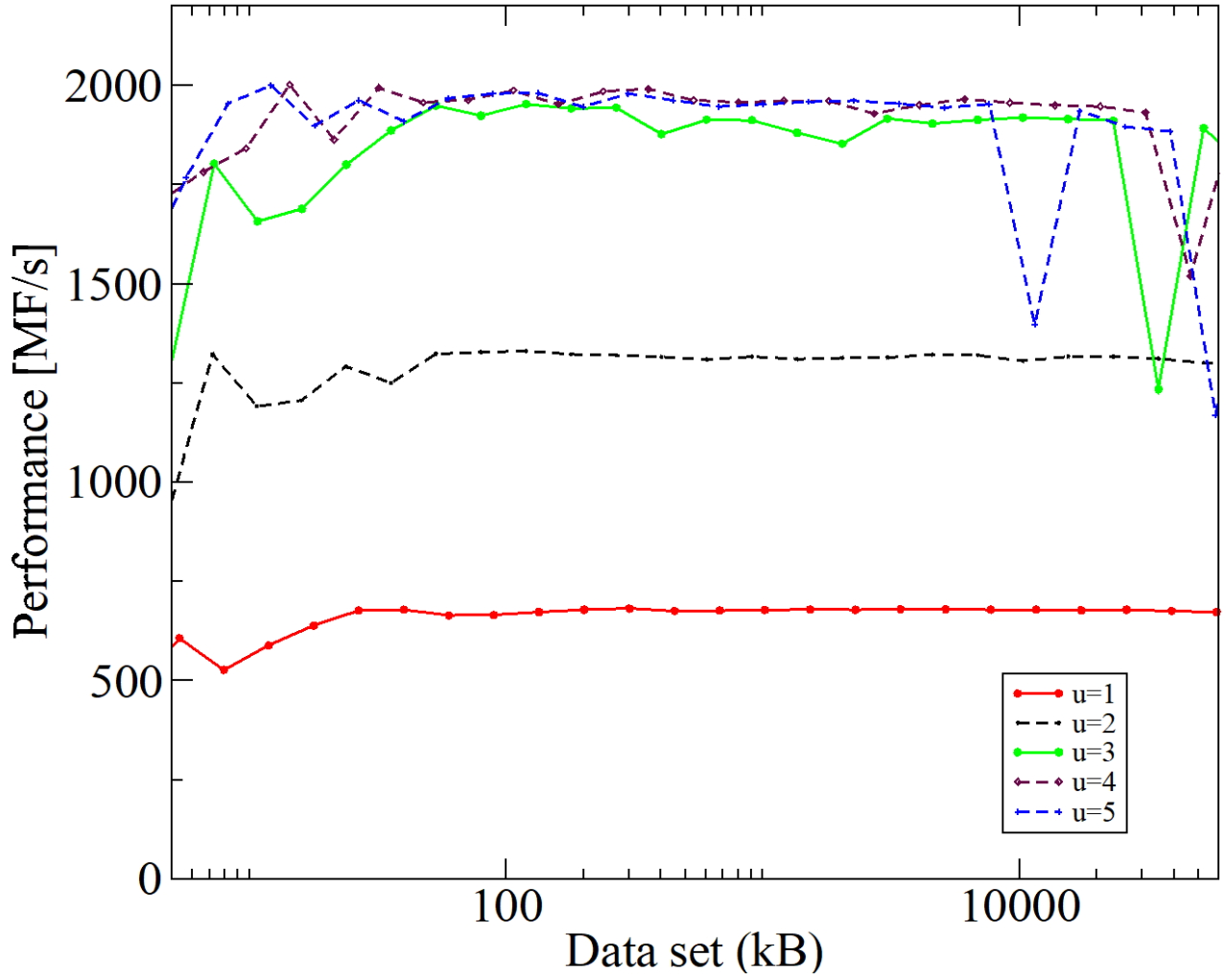
u=2 Dependencies

```
do i=2,N
   A(i) = A(i-1) * s
   B(i) = B(i-1) * s
end do
```

Latency/depth of MULT pipeline?

Pipelining: Data dependencies

- Single core on AMD EPYC 7543 CPU with clock speed fixed to **2.1 GHz** (Compiler: `-O3 -no-vec`) – HW limit: 2 MULT instr./cy



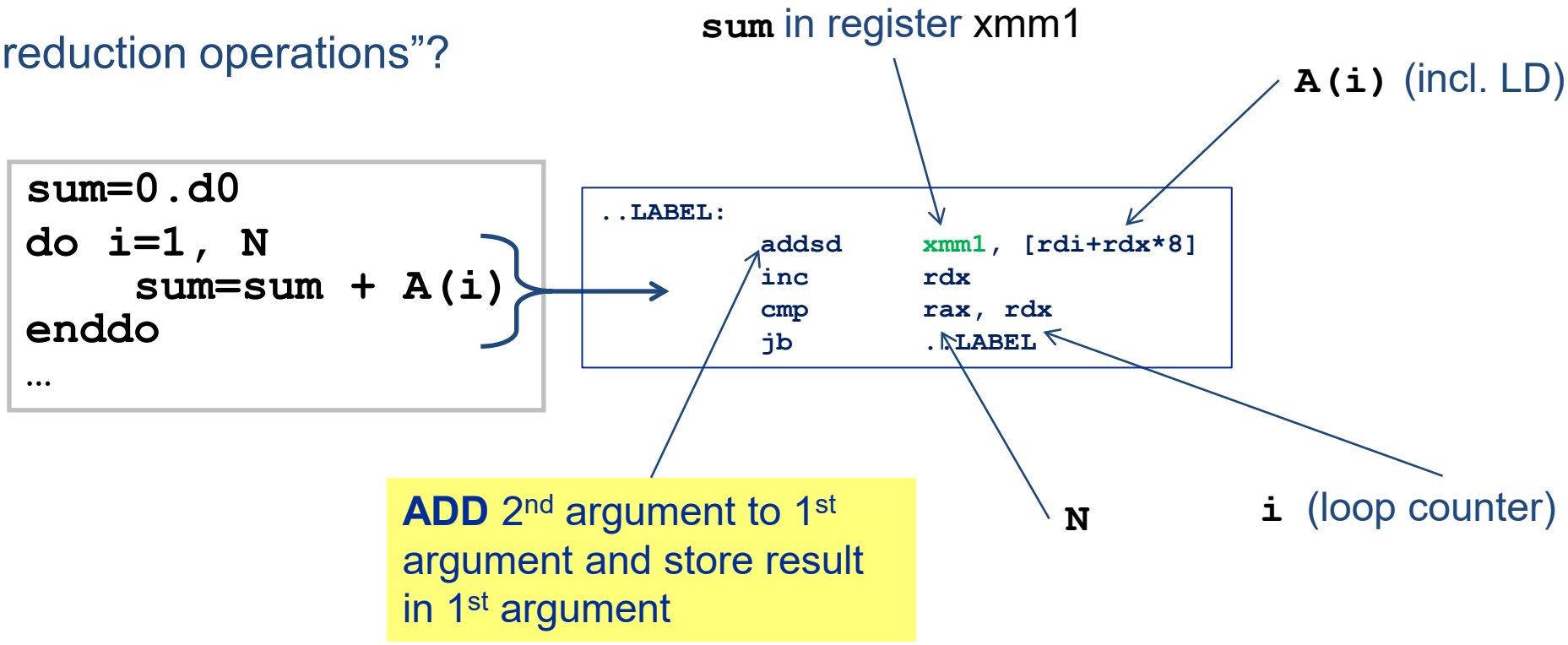
u=2 Dependencies

```
do i=2,N
   A(i) = A(i-1) * s
   B(i) = B(i-1) * s
end do
```

Latency/depth of MULT pipeline?

Pipelining: Resolving dependencies

- Sometime the data dependencies are not that obvious..
- What about “reduction operations”?



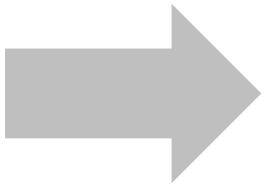
- Data (register) dependency on **sum** (xmm1) → 1 F / m cy for above code! (assuming an ADD latency of m cycles, m=3 for Intel)
- How to enable pipelining here?

Pipelining: Resolving dependencies

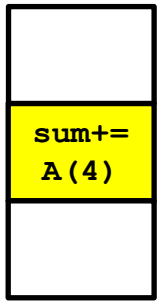
- Increase pipeline utilization by “loop unrolling”

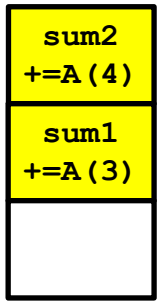
“2-way Modulo Variable Expansion” (we assume that N is even)

```
sum=0.d0
do i=1, N
   sum=sum+A(i)
enddo
```



```
sum1=0.d0
sum2=0.d0
do i=1, N, 2
   sum1=sum1+A(i)
   sum2=sum2+A(i+1)
enddo
sum = sum1 + sum2
```





Pipelining: Resolving dependencies

- m-way Modulo Variable Expansion (MVE) to get best performance!
- Sum is split up in m independent partial sums
- Optimal for Intel ADD: 3-way MVE

```
Nr=3*(N/3)
sum1=0.d0
sum2=0.d0
sum3=0.d0
do i=1, Nr, 3
   sum1=sum1+A(i)
   sum2=sum2+A(i+1)
   sum3=sum3+A(i+2)
enddo
do i=Nr+1, N
   sum1=sum1+A(i)
enddo
sum=sum1+sum2+sum3
```

Remainder loop

Pipelining: Resolving dependencies

- Compiler can do that, if it is allowed to do so…
  - High optimization levels
  - Compiler prefers powers of 2 for unrolling

- Reason: Computer's floating point arithmetic is not associative!

$$\Big(\big((a+b)+c\big)+d\Big)+e\big)+f\Big) \neq (a+b)+(c+d)+(e+f)$$

- If you require binary exact results (`-fp-model strict` for Intel) the compiler is not allowed to do this transformation

- Beware additional latency due to reduction at the end
  - Final sum cannot be pipelined
  - High unrolling factor leads to high overhead
  - High unrolling may lead to register shortage

Pipelining: Available resources in modern CPUs

- Typical number of pipeline stages on modern cores:
  - 2-5 for most (important) hardware pipelines: **LoaD**; **STore**; **MULT**; **ADD; FMA**
  - >>10 for other floating point pipelines: **DIV**ide/**SQ**uare**R**oo**T**
  - Many other other piplined ALUs, e.g. integer arithmetic, logical, shift, branch, address generation

- Most “older” x86 cores (AMD, Intel):
  - 1 MULT & 1 ADD floating point unit per processor core
    → Max. 1 MULT & 1 ADD instruction per cycle

- Latest Intel (Haswell, Broadwell, Skylake) & AMD (Zen+) cores:
  1 (AMD) or 2 Floating Point Fused MultiplyAdd (FMA) floating point units
  - FMA3 instruction: s=s+a*b → 1 Input register (s) is overwritten
  - FMA4 instruction: s=r+a*b → No input register is modified
  - Typically 2 (1) FMA instruction per cycle for Intel (AMD) processors
  - On Intel: Per cycle up to 2 MULT or ADD instructions

Costs of arithmetic instructions: Intel Skylake processors

Latency [cy/instruction]
Depth of pipeline, i.e. cycles to execute a single instruction (worst case)

Throughput [instruction/cy]
Cycles per instruction if pipeline is full (best case: 2 instruction/cy – 2 HW units)

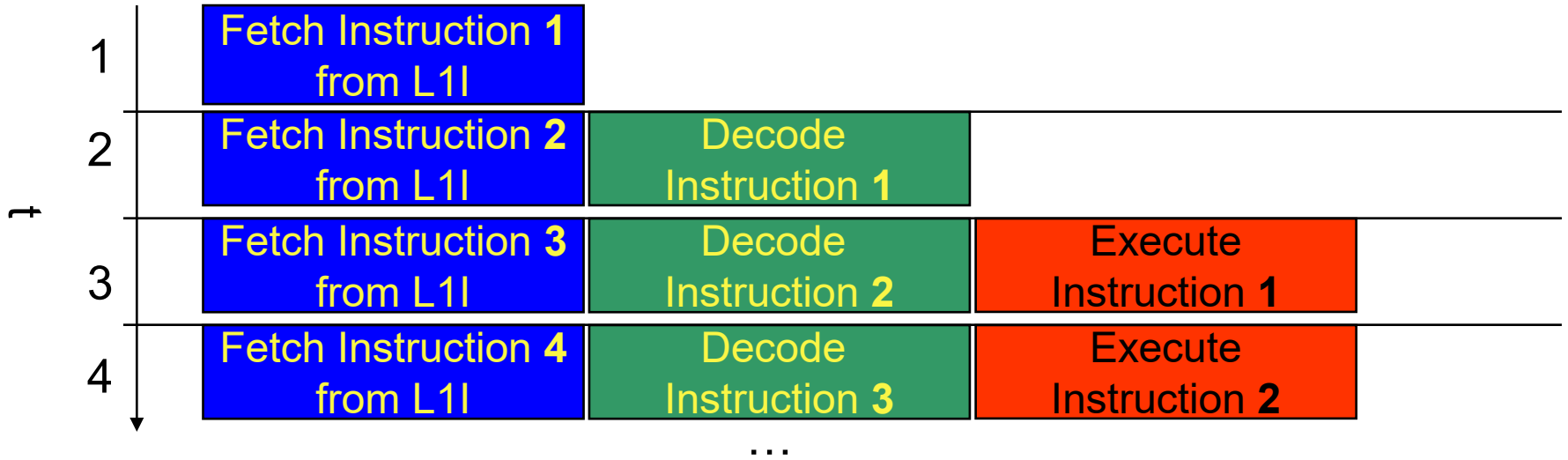
	Instruction	Latency [cy/instruction]	Max. throughput [instructions/cy]
Processor (scalar) instructions	ADD DP (SP)	4 (4)	2 (2)
	MULT DP (SP)	4 (4)	2 (2)
	FMA DP (SP)	4 (4)	2 (2)
	SQRT DP (sqrtsd)	26	1/12 = 0.08
	SQRT SP (sqrtss)	20	1/6 = 0.16
	DIV DP (divsd)	14	1/4 = 0.25

- Consequence: Avoid expensive instructions in hot spots!
- Other expensive math (transcendental, log,…) is done in libraries

Pipelining: The Instruction pipeline

- Besides arithmetic & functional units, instruction execution itself is pipelined also, e.g.: one instruction performs at least 3 steps:

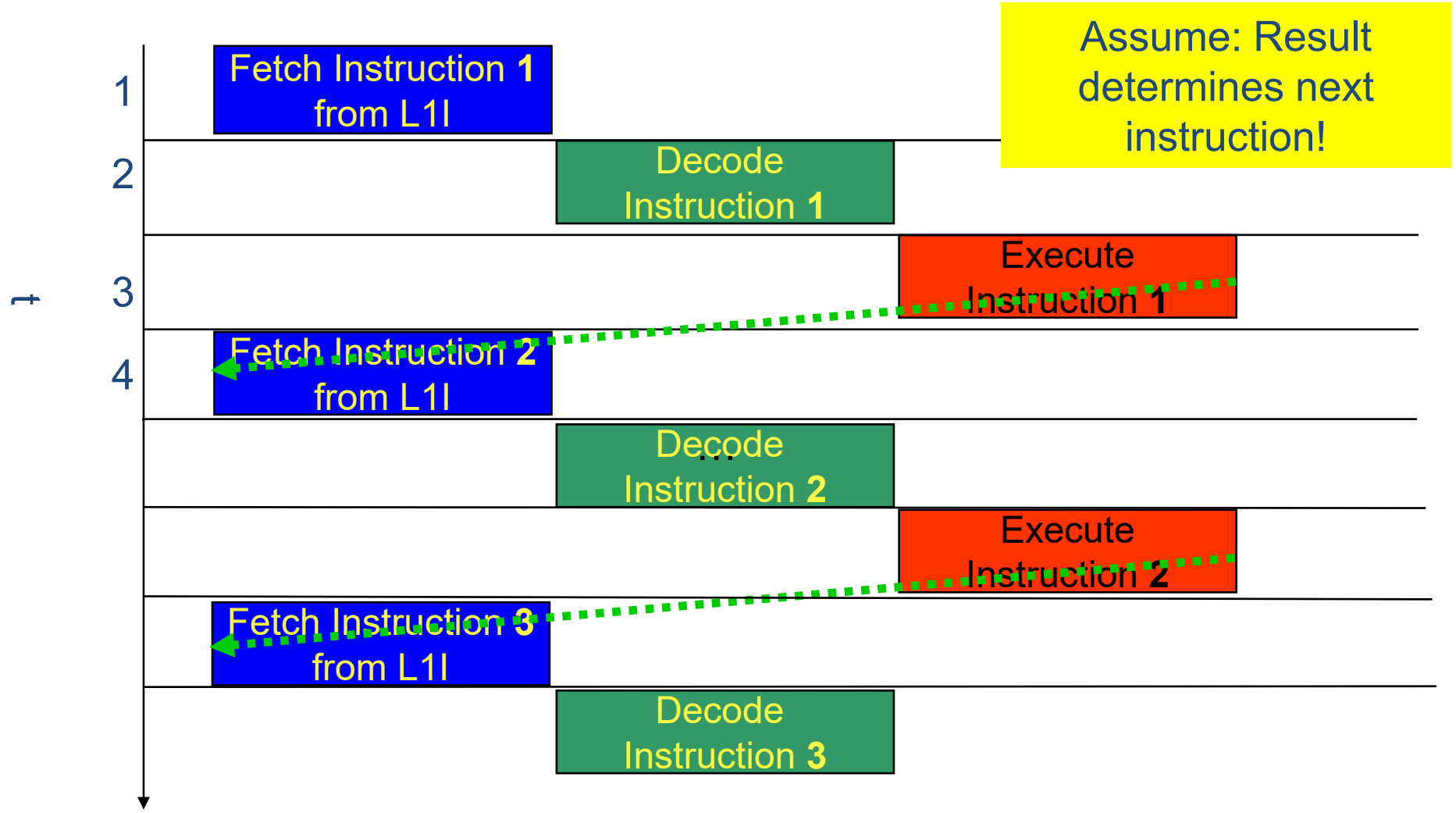
- Hardware Pipelining on processor (all units can run concurrently):



- Non-predictable branches can stall this pipeline!
  - Hardware can predict conditional branches w/ high accuracy
- Each unit is pipelined itself (cf. Execute=Multiply Pipeline)

Pipelining: The Instruction pipeline

- Problem: Unpredictable branches to other instructions



Pipelining summary

- Pipelining tries to achieve
  - Maximum instruction throughput (1 instr/cy in many cases)
  - Hiding of instruction latency

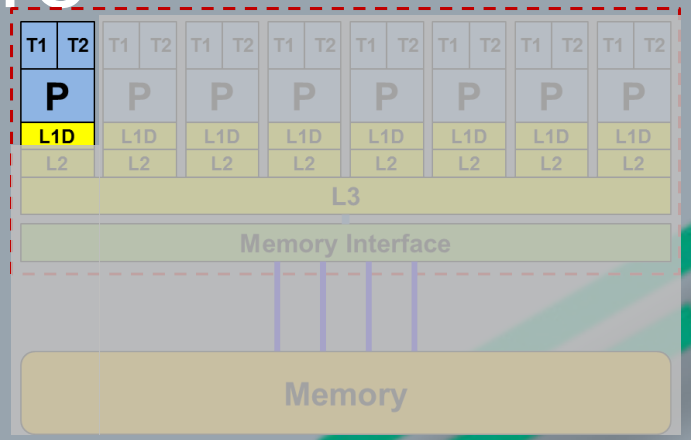
- Prerequisites
  - Independent instructions
  - *A lot of* independent instructions for maximum efficiency ($N \gg m$)
  - Highest benefit if code & data are close to the core (L1 instr./data cache)
  - Conditional branches must be correctly predicted by hardware

- Drawbacks
  - Pipeline must be filled → inefficient for $\mathrm{N} \lesssim m$
  - Dependencies between pipelines may increase effective depth (see tutorial)
  - Unresolvable data dependencies are hazardous

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Programming Techniques for Supercomputers
Modern processors: Single Core



- Introduction
  - Basic technology trend / Moore's law
  - Basic concept of core architecture
- Key single core features:
  - Pipelining
  - Superscalarity
  - SingleInstructionMultipleData
- Maximum In-Core Performance

Superscalar Processors

- Superscalar processors provide additional hardware (i.e. transistors) to execute multiple instructions per cycle!

  → Exploit Instrucion Level Parallelism (ILP)

- Parallel hardware components / pipelines are available to
  - fetch / decode / issues multiple instructions per cycle (typically 3 – 8 per cycle)
  - perform multiple integer / address calculations per cycle
  - perform multiple load (store) multiple instructions per cycle (e.g. one LD and one ST per cycle)
  - perform multiple floating point (FP) instructions per cycle (e.g., 2 floating point instructions/cycle, e.g. 1 MULT + 1 ADD)

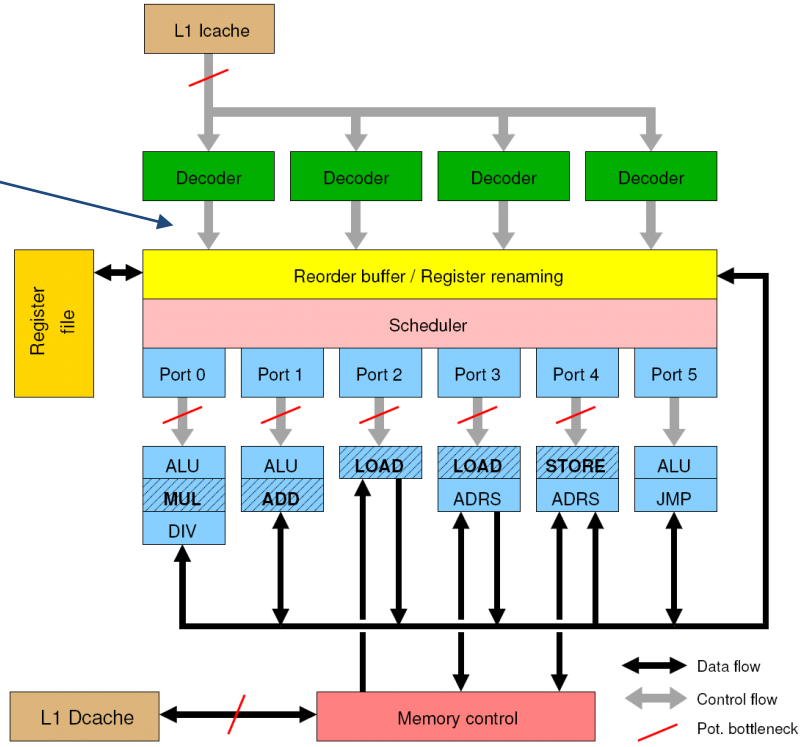
- “Parallelization of instruction stream” required

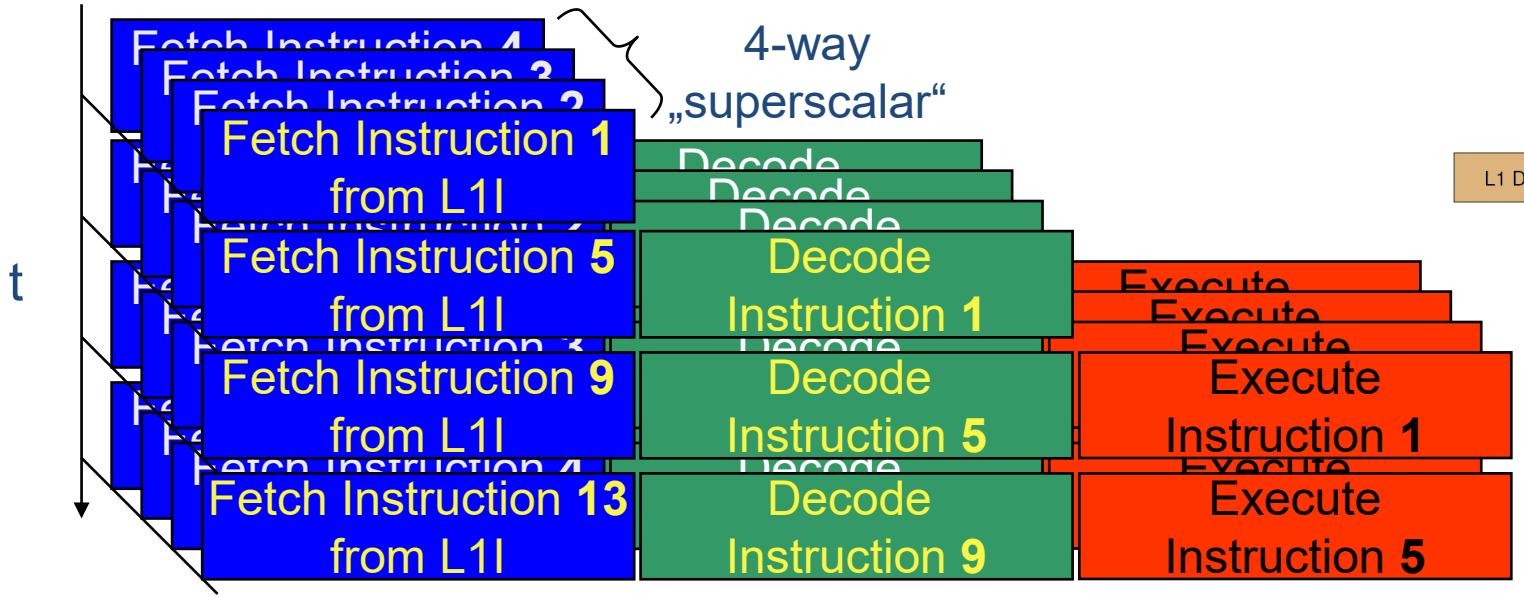
- Performance metrics quantifying superscalarity:

  Instructions Per Cycle: IPC
  Cycles Per Instruction: CPI

Superscalar Processors – Instruction Level Parallelism

- Issuing m concurrent instructions per cycle:
  "m-way superscalar"
- Modern processors are 3- to 8-way superscalar
  & perform 2 or 4 FP instructions per cycles





Multiple pipelines at work: Interleaving instructions

- Example:

Fortran Code:

```
do i=1,N
   a(i) = a(i) * c
end do
```

```
load r1, a[i]
mult r1 = c,r1
store a[i], r1
branch.loop
```

Assumed Latencies

Load operand to register (4 cycles)
Multiply a(i) with c (2 cycles); a[i],c in registers
Store result from register to mem./cache (2 cycles)
Increase loop counter as long as i less or equal N (0 cycles)

Simple Pseudo Code:

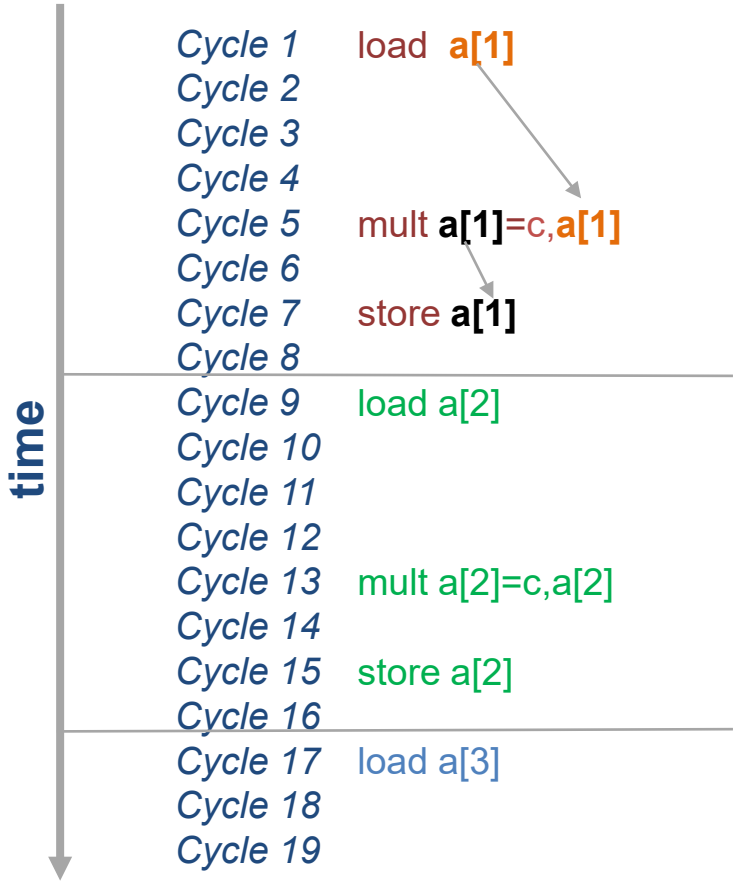
```
loop:  load r1, a[i]
       mult r1 = c, r1
       store a[i], r1
       branch.loop
```

- Dependencies on **r1**
  - within one iteration
  - across iterations

Superscalar & Pipelined Execution

`a[i]=a[i]*c`; N=12

Naive instruction issue



Simple Pseudo Code:

```
loop:   load a[i]
        mult a[i] = c, a[i]
        store a[i]
        branch.loop
```

Instruction executed „**in-order**“

Total execution time:

T= 12 * (4+2+2) cy = 96 cy

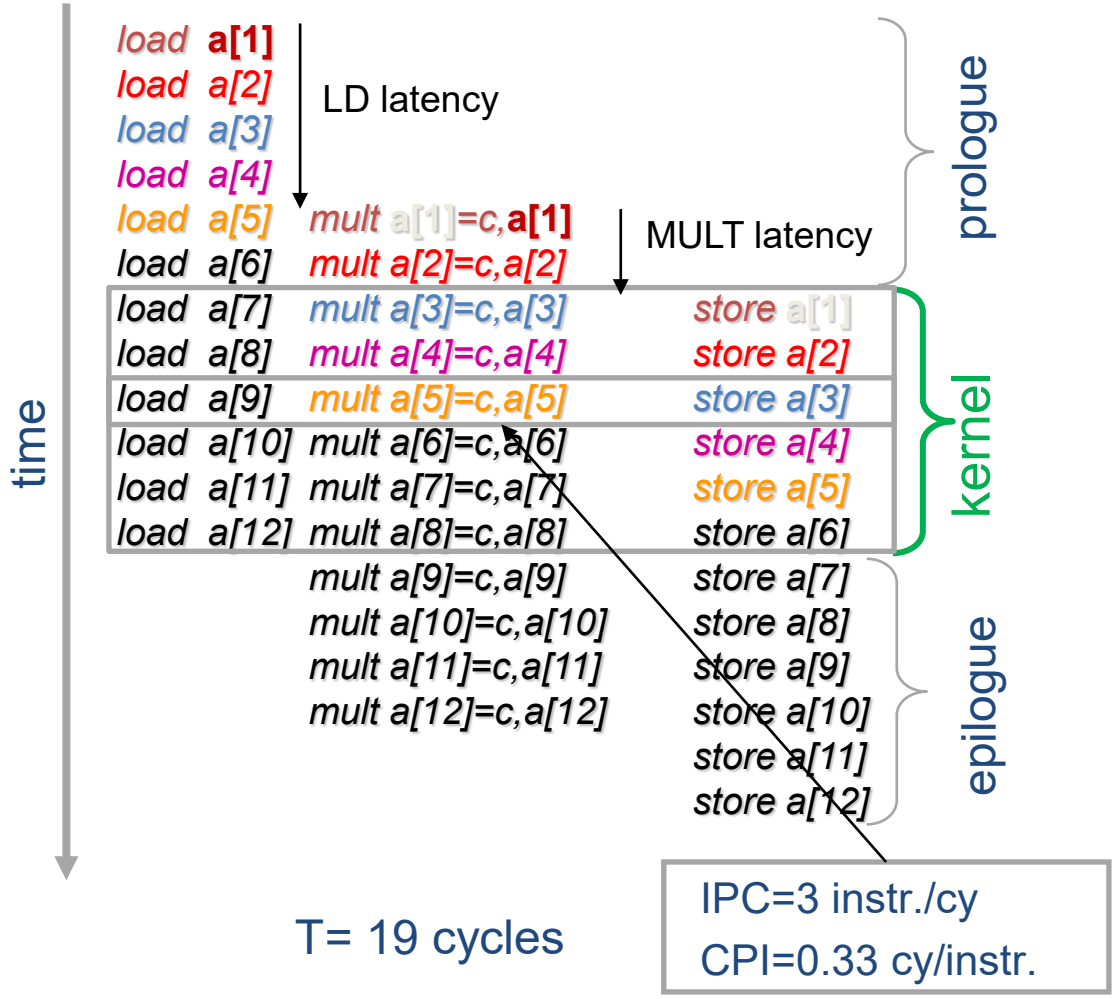
IPC = 3/8 instr./cy
CPI = 8/3 cy/instr.

No pipelining and superscalarity!

Superscalar & Pipelined Execution

`a[i]=a[i]*c`; N=12

Optimized instruction issue



T= 19 cycles

IPC=3 instr./cy
CPI=0.33 cy/instr.

Simple Pseudo Code:

```
loop:   load a[i]
        mult a[i] = c, a[i]
        store a[i]
        branch.loop
```

Assumptions:

- LD/MULT/ST can be executed in parallel!
- Instructions are perfectly reordered but dependecies (within loop iteration) are maintained!
- Register renaming required

Kernel:
Full pipelining and high superscalarity!

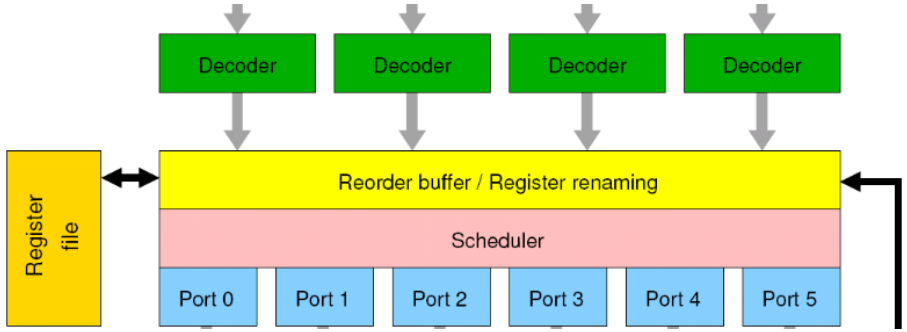
Reordering the instruction stream: Two options

- Software pipelining
  - Done by the compiler
  - Compiler reorders instructions
  - Requires deep insight into application (data dependencies) and processor (latencies of functional units)

```
<… prologue …>
kernel:  load a[i+6]
         mult a[i+2] = c, a[i+2]
         store a[i]
         branch → kernel
<… epilogue …>
```

  - Required on “in-order” architectures
  - Rarely used today (see right)

- Dynamic reordering of instructions at runtime
  - Done by the hardware
  - Out-of-order (OOO) execution
  - Instructions are executed when operands are available



  - All modern general-purpose CPUs do this

Register renaming

- Prerequisite for good OoO execution: “Bogus” register dependencies can be resolved
  - Hardware has “shadow registers” it can use to store intermediate values that are already “officially” overwritten

```
for(int i=1; i<n; ++i)
        a[i] = a[i] + s;
```

```
LOOP:
     LOAD r1 = a[i]
     ADD r1 = r1+r2
     STORE a[i] = r1
     i++
     i<n ? BRANCH : EXIT
```

This looks like a dependency: How can iterations overlap if they need the same register `r1`?

  - Solution: Hardware assigns a new register with the same name as soon as the old value gets overwritten
  - “Shadow copy” lives as long as necessary
    - Until no instructions in flight reference the register any more

Superscalar processors
Intel processors – qualitative view (“Intel Sandy Bridge”)