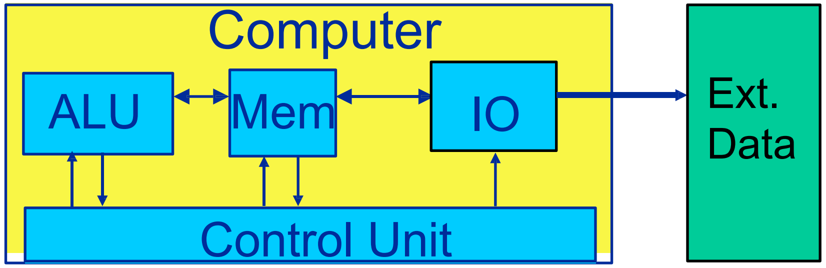
Single Node Performance

CPSC 424/524
Lecture #2
August 31, 2018

<u>Credit</u>: Almost all of these slides are courtesy of Prof. Gerhard Wellein, who developed them for use in HPC programming courses at University of Erlangen, Germany

Stored-Program Computer Architecture

- Modern computer architectures still implement the stored-program architecture [Turing (1936), EDVAC (1949)] widely known as the *von Neumann architecture*



Control unit: Fetches and processes low-level instructions from memory

Arithmetic/Logic Unit (ALU): Performs computations/manipulations of data stored in memory (controlled by control unit)

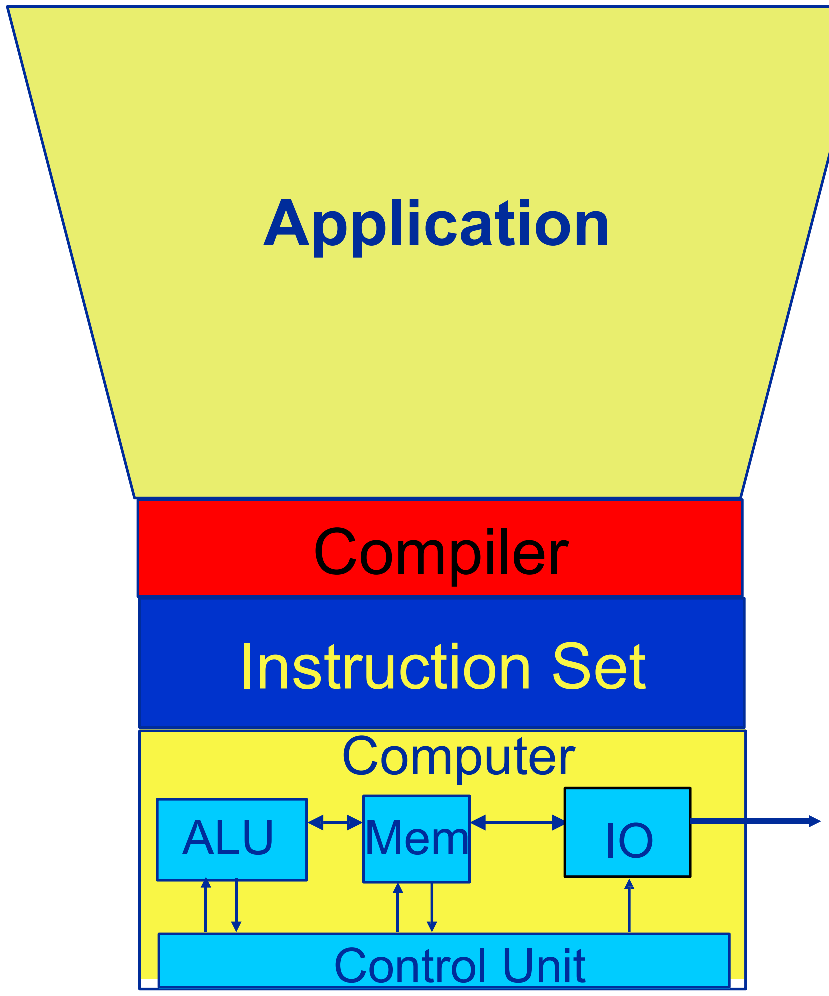
(CPU = Control unit + ALU)

IO: Communication & external data access

Very quickly several major issues were identified:

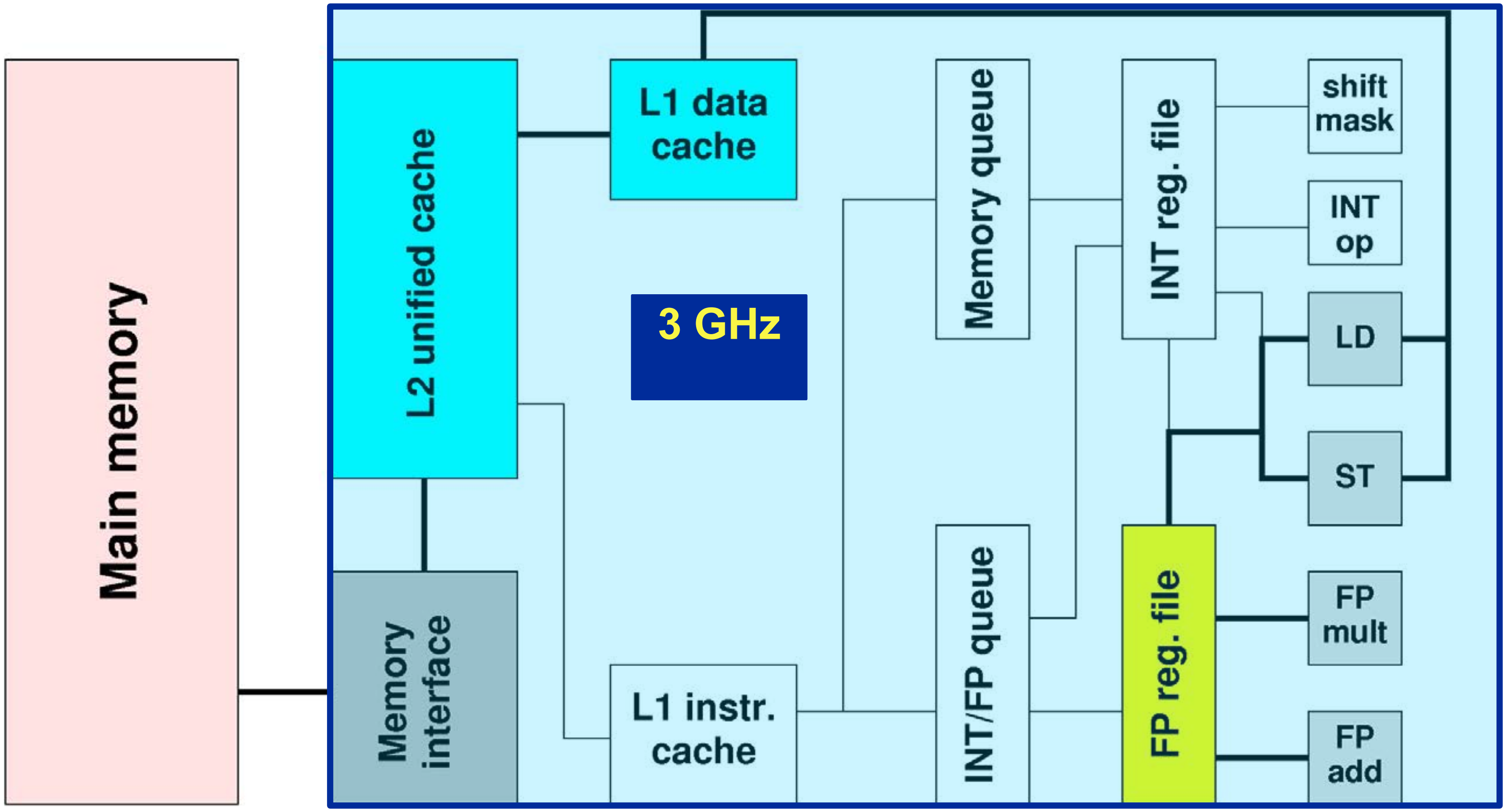
- ***von Neumann bottleneck***: performance is limited by memory access speed
- Inherently sequential (single instruction working on a few data items)
- Original implementations were difficult to program (low-level instructions sets)

More Modern View



- **Application: "Portable”; High-Level Programming Language (e.g. C / C++ / Fortran)**
- **Compiler translates program to machine-specific instruction set**
- **Stored program/von Neumann concept is still visible, but**
  - Several memory levels: Register – L1 – L2 – L3 – main memory
  - Multiple arithmetic/logical units (e.g., 2 floating point units for Core i7)

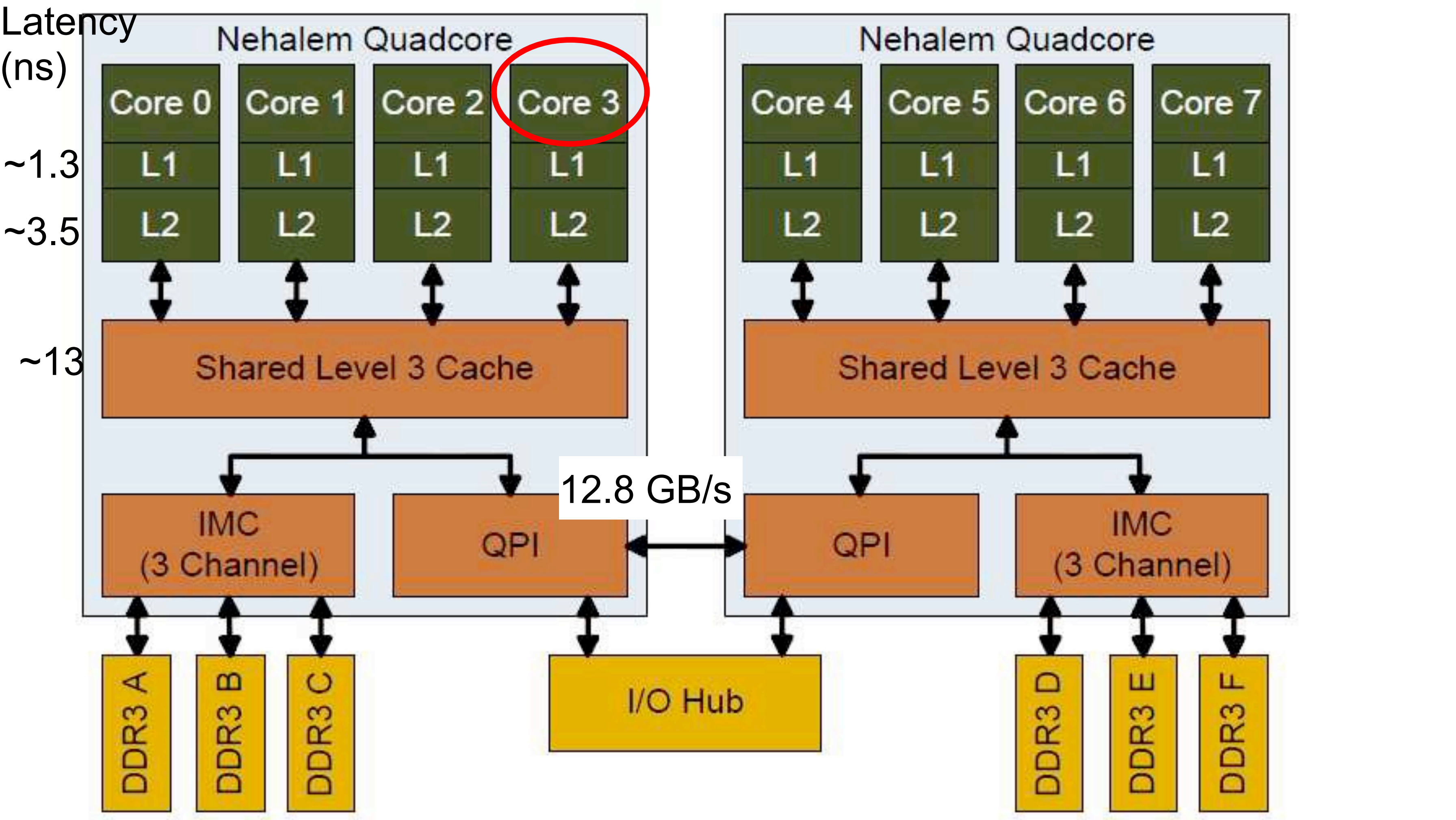
Modern general-purpose cache based microprocessor



Today, some memory (caches) are integrated on the processor chip
Not shown: most of the control unit, e.g. instruction fetch/decode, branch prediction,…

Courtesy of Prof. G. Wellein, U. of Erlangen

Nehalem (Core i7) Memory Hierarchy



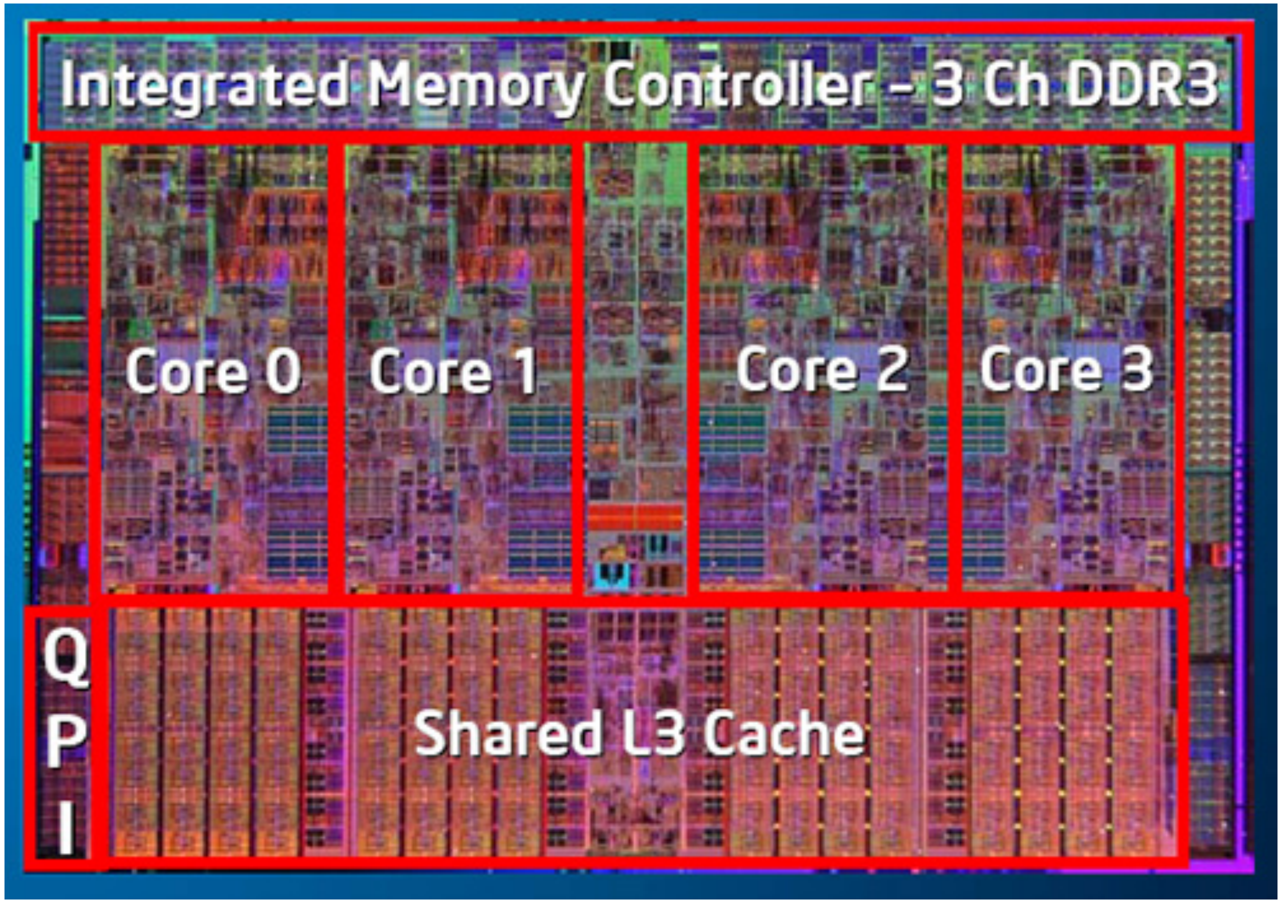
65 ns local latency
Local BW: 8 GB/s/core

NUMA: Non-uniform memory access

105 ns remote latency

From T. Rolf, "Cache Organization and Memory Management of the Intel Nehalem Computer Architecture"

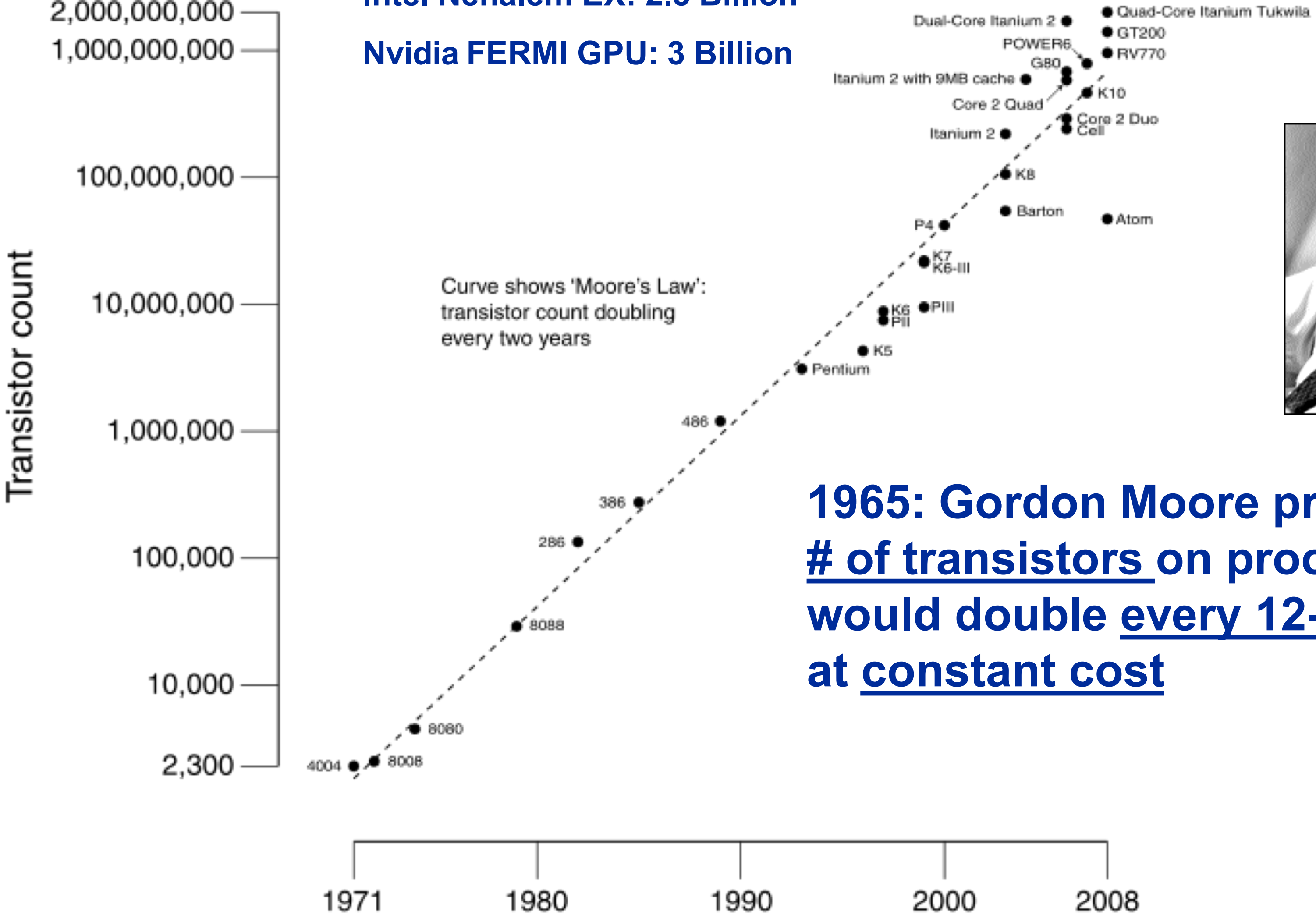
Nehalem Processor Chip

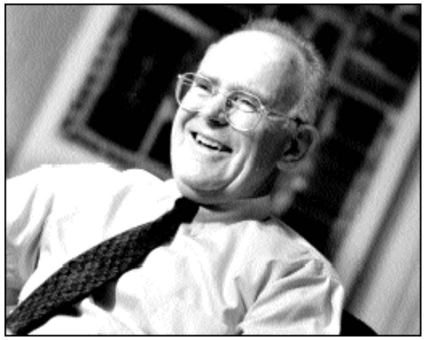


Moore's law

Intel Nehalem EX: 2.3 Billion

Nvidia FERMI GPU: 3 Billion

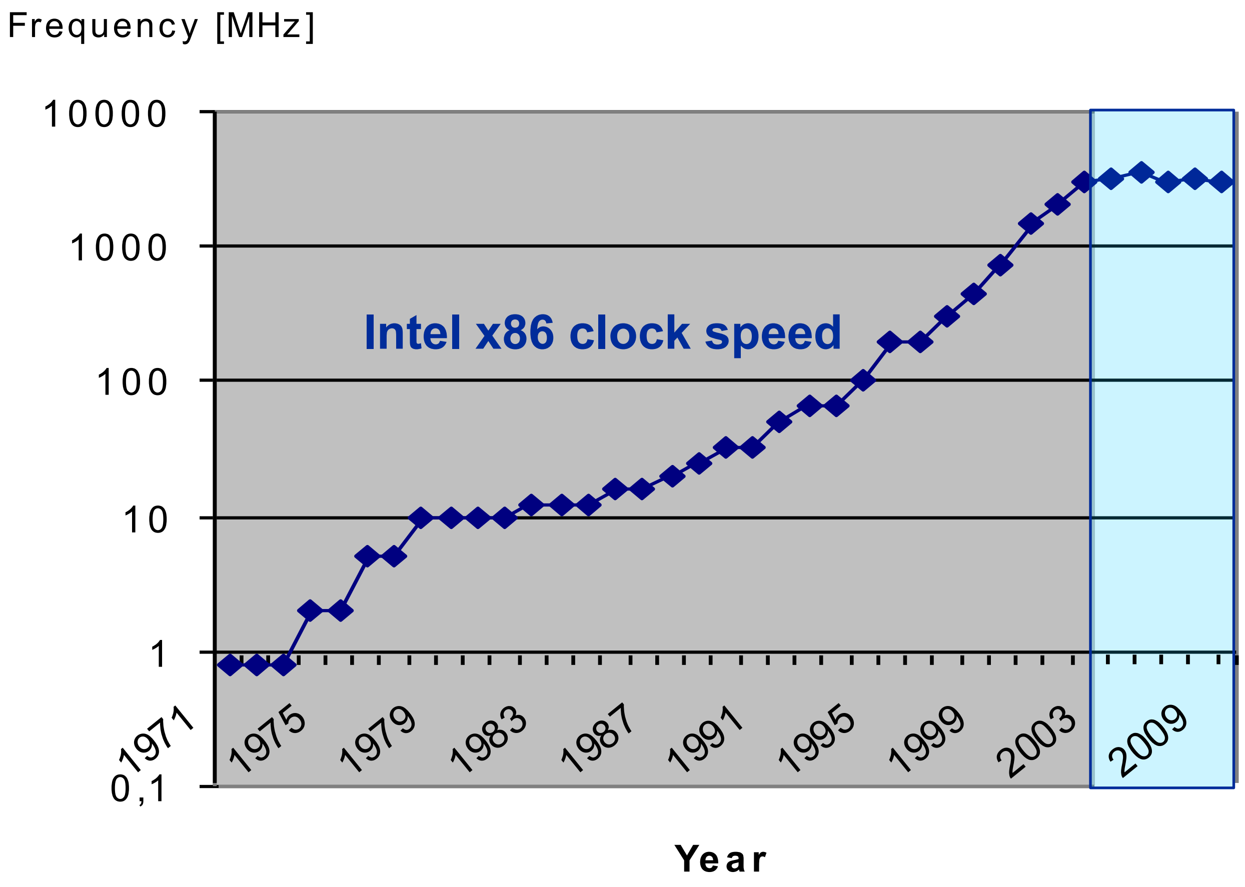




1965: Gordon Moore predicted that <u># of transistors</u> on processor chip would double <u>every 12-24 months</u> at <u>constant cost</u>

Moore's law → faster cycles and beyond

- Moore's law → smaller transistors → faster clock speeds
- Faster clock speeds → Higher Throughput (Ops/sec)
- But, ultimately, clock speed is limited by power concerns



Increasing transistor count and limited clock speed allows & forces architectural changes:

- Pipelining
- Superscalarity
- SIMD / Vector ops (AVX, etc.)
- Multi-Core/Threading
- Complex on chip caches

Courtesy of Prof. G. Wellein, U. of Erlangen

Instruction Set Paradigms

- **In the 60's: Complex Instruction Set Computers (CISC):**
  - Powerful & complex instructions directly reflecting the hardware
  - Instructions could perform several (independent) operations, enabling some parallelism within instructions
  - Instructions took varying amounts of time and space
  - Difficult to build compilers
- **Mid 80's: Reduced Instruction Set Computer (RISC):**
  - Simplified instructions: single operations taking fixed numbers of cycles
  - Designed for fast clocks, many registers, caches, and "pipelining"
  - Targetable by highly optimizing compilers
  - Complex operations split into simple steps:

    e.g.: A=B*C is split into at least 4 operations, as in

    ```
    LD B→r0; LD C→r1; MULT r0*r1→r2; ST r2→A
    ```
- **Now: Superscalar RISC processors**
  - Processors contain multiple execution units for parallelism
  - Complex chip designs that try to gain performance from replication, not necessarily clock speeds.

Pipelining of arithmetic/functional units

- **Idea: Assembly Line**
  - Split complex instructions into several simple / fast steps (stages)
  - Each step takes the same amount of time (e.g. a single cycle)
  - Execute different steps from different instructions at the same time (in parallel)
- **Allows for shorter cycle times (simpler logic circuits), e.g.:**
  - Floating point multiplication takes 5 cycles, but …
  - Processor can work on 5 different multiplications simultaneously
  - So: can deliver one result per cycle after the pipeline is full
- **Drawbacks:**
  - For efficiency: many independent, identical instructions (e.g., scaling a vector)
  - Pipeline must be filled (“wind up” time) (Want instruction count >> pipeline steps)
  - Requires complex instruction scheduling by compiler & hardware – software-pipelining, out-of-order execution, etc.
- **Pipelining is widely used in modern computer architectures**

Example: Possible Stages for Floating Point Multiplication

- **Floating point numbers are represented as a sign, a "normalized" mantissa, and an exponent:** `s*0.m * 2`^{e} **with**
  - **Sign** `s ϵ {-1,1}`
  - **Mantissa** `m` **which does not contain 0 as leading bit**
  - **Exponent** `e` **some positive or negative integer**

  `m` **&** `e` **come in two lengths (single & double precision)**

- **Multiply two FP numbers in registers:** `r1*r2` $\longrightarrow$ `r3`

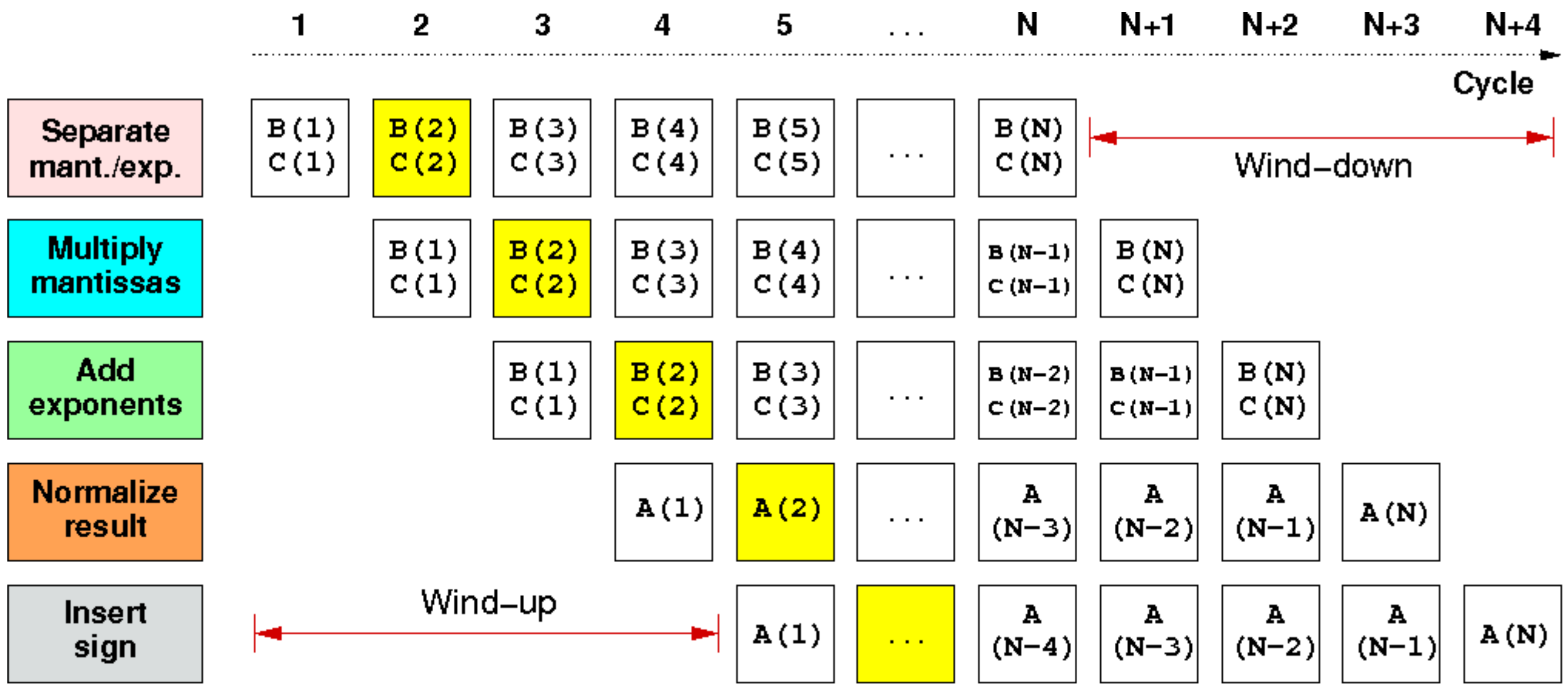
  `r1=s1*0.m1 * 2`^{e1} , `r2=s2*0.m2 * 2`^{e2}

  `s1*0.m1 * 2`^{e1} `*` `s2*0.m2 * 2`^{e2}

$\rightarrow$ $(s1*s2)*\ \underbrace{(0.m1*0.m2)\ *\ 2^{(e1+e2)}}$

$\rightarrow$ **Normalize result:** $s3*\ \overbrace{0.m3\ *\ 2^{e3}}$

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



Wind-up/-down phases: Empty pipeline stages

Pipelining: Speed-Up and Throughput

- Assume a general m-stage pipeline, i.e. pipeline depth is m.
- <u>Speed-up</u>: pipelined vs non-pipelined execution at same clock speed for N ops. (Note: "time" is represented here by cycles.)

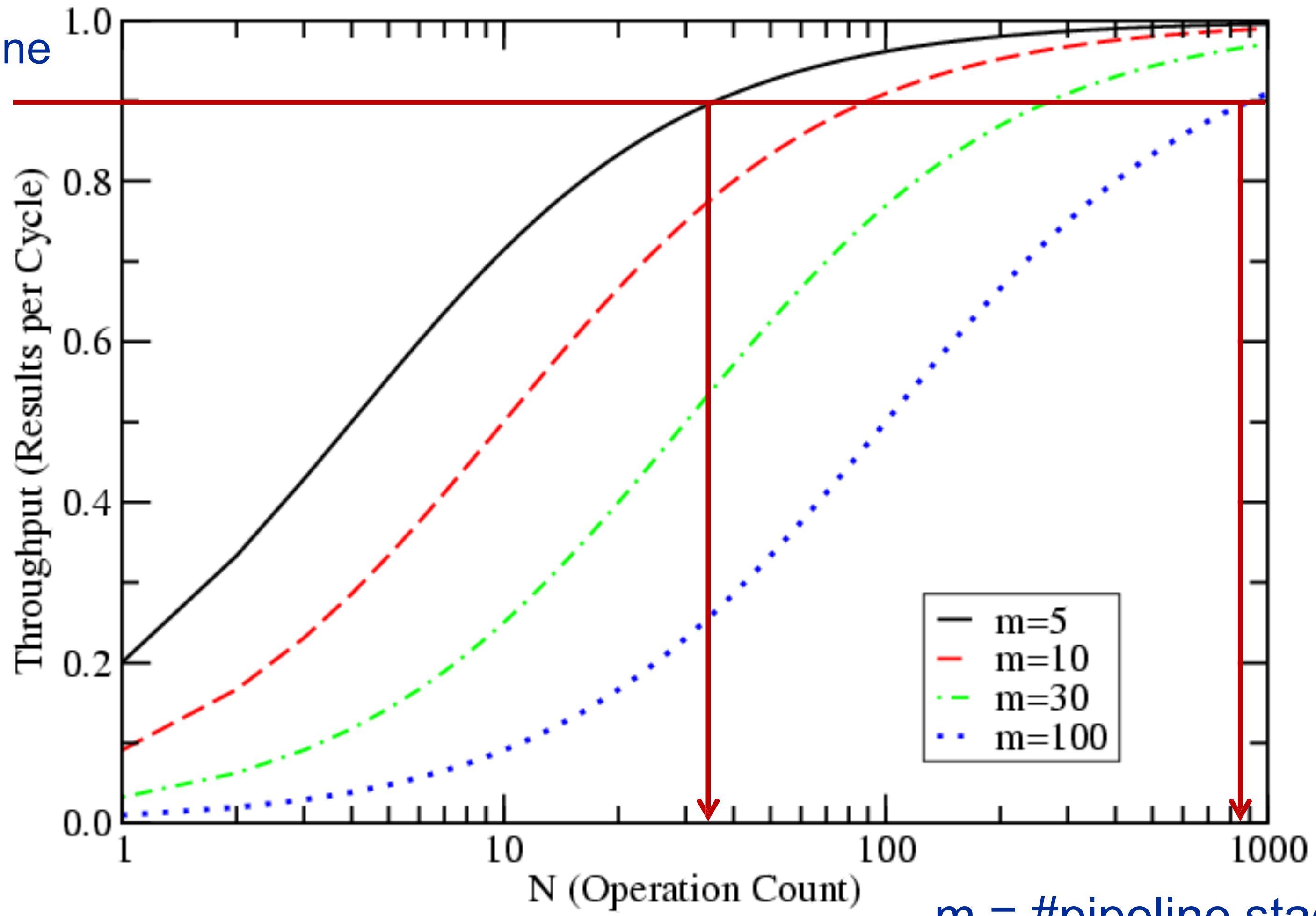
$$T_{seq} / T_{pipe} = (m*N) / (N+m) \approx m \text{ for large } N\ (>>m)$$

- <u>Throughput</u> of pipelined execution (= Average # results per Cycle) executing N instructions in pipeline with m stages:

$$N / T_{pipe}(N) = N / (N+m) \approx 1 \text{ for large } N$$

Throughput as function of pipeline stages

90% pipeline efficiency

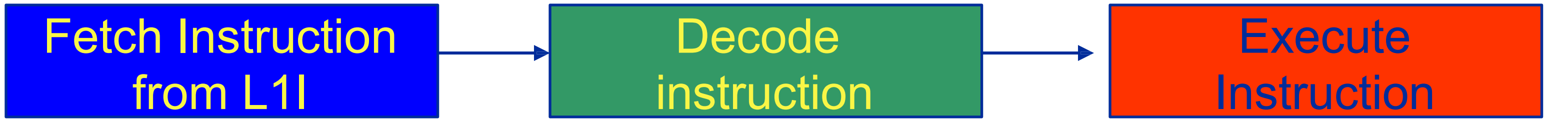


m = #pipeline stages

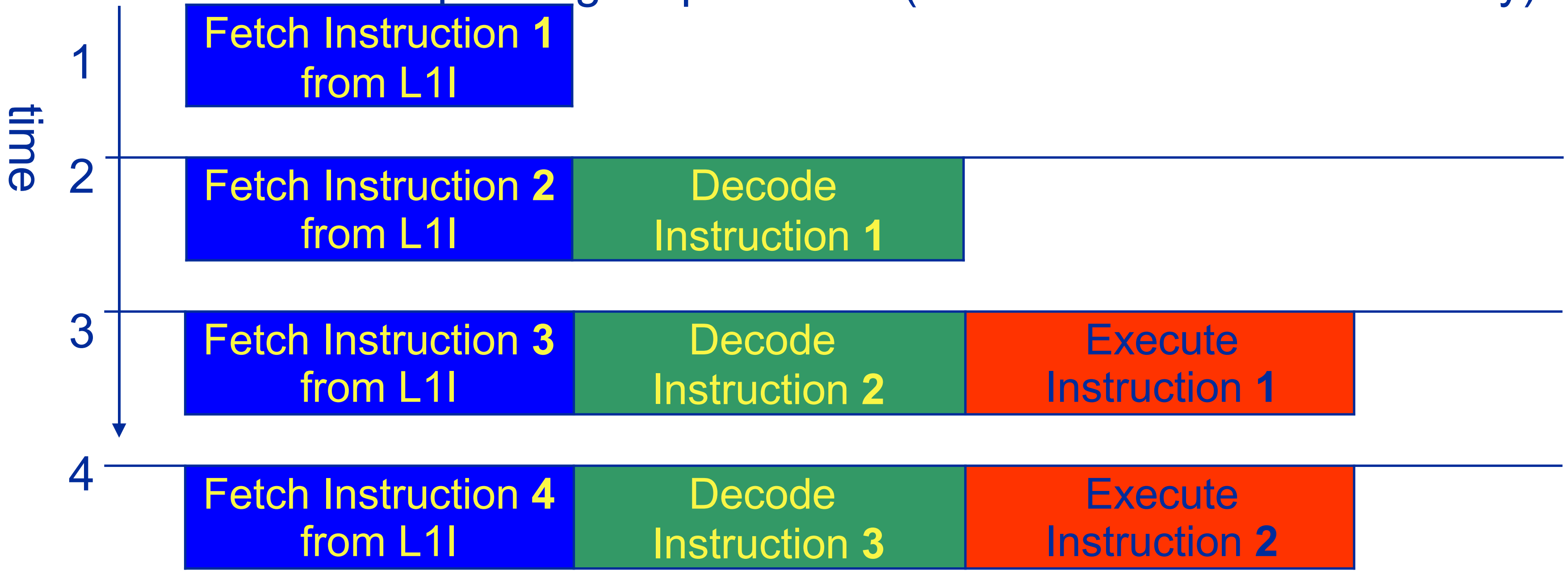
Courtesy of Prof. G. Wellein, U. of Erlangen

Pipelining: The Instruction pipeline

- **Instruction execution is pipelined; that is each instruction requires at least 3 substeps:**



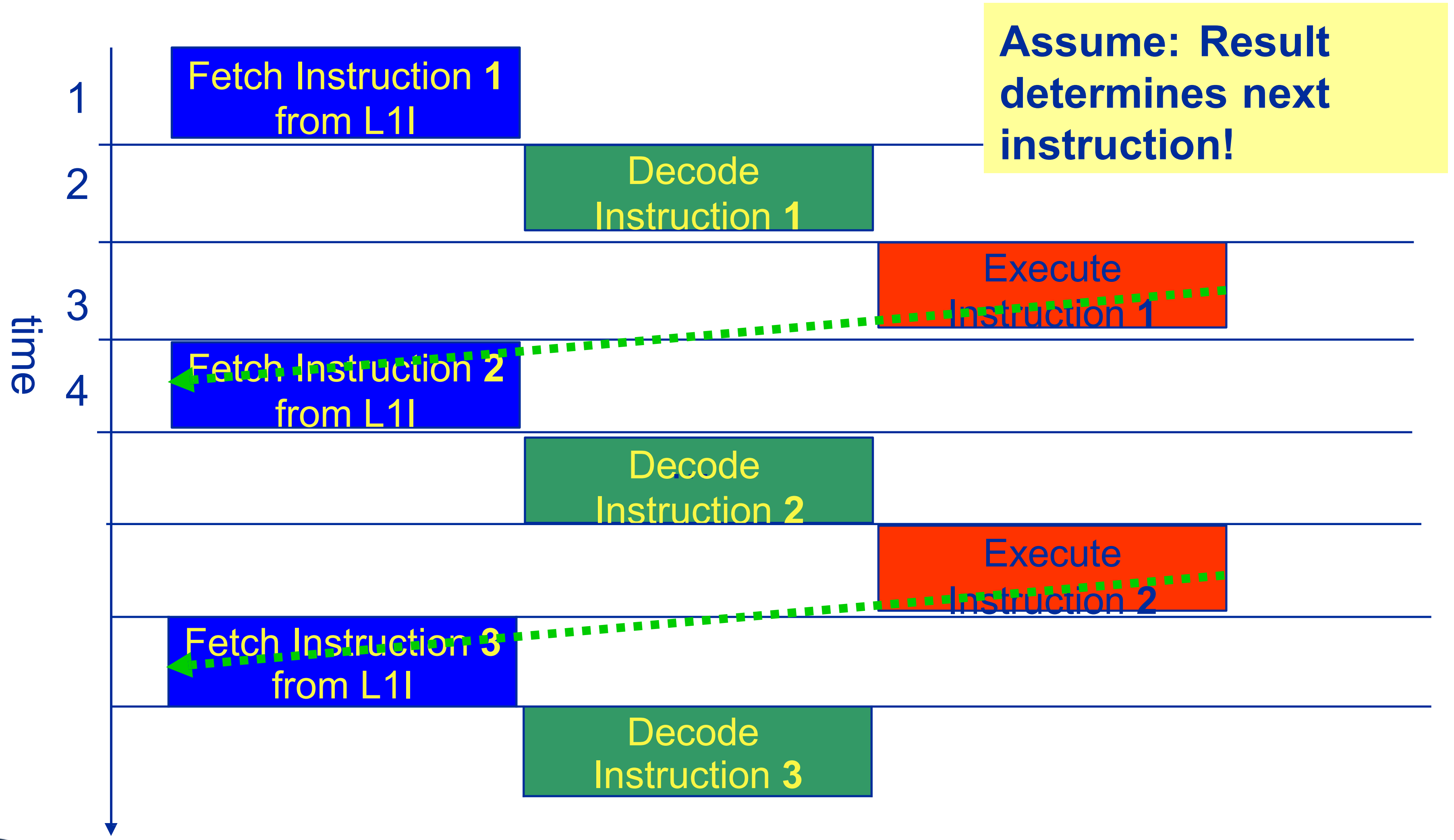
- Hardware Pipelining on processor (all units can run concurrently):


time
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

- Each Unit is pipelined. ("Execute" may be the Multiply Pipeline)
- Branches can stall this pipeline! (Speculative Execution, Prediction)

Pipelining: The Instruction pipeline

- **Problem: Unpredictable branches to other instructions**



Assume: Result determines next instruction!
time
1
2
3
4
Fetch Instruction 1 from L1I
Decode Instruction 1
Execute Instruction 1
Fetch Instruction 2 from L1I
Decode Instruction 2
Execute Instruction 2
Fetch Instruction 3 from L1I
Decode Instruction 3

Superscalar Processors

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

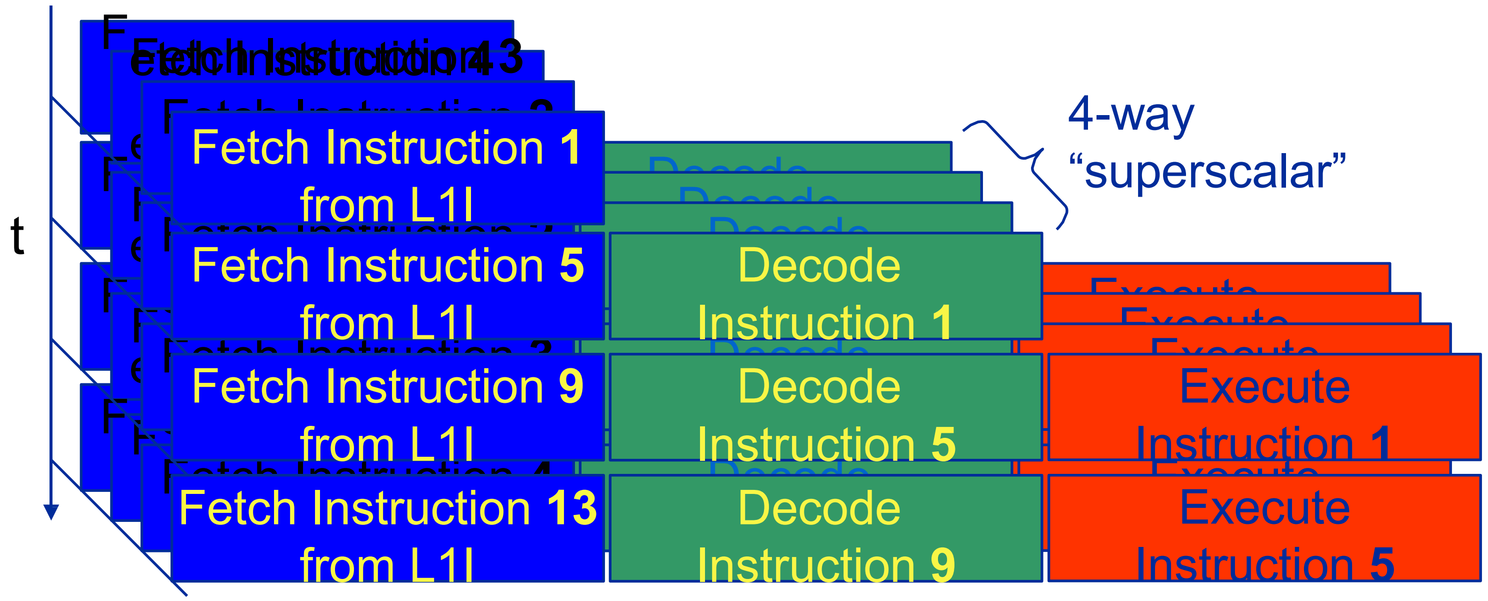
- **Parallel hardware components / pipelines are available to**
  - fetch / decode / issues multiple instructions per cycle (typically 3 – 6 per cycle)
  - perform multiple integer / address calculations per cycle (e.g. 6 integer units on Itanium2)
  - load (store) multiple operands (results) from (to) cache per cycle (typically one load AND one store per cycle)
  - perform multiple floating point instructions per cycle (typically 2 floating point instructions per cycle, e.g. 1 MULT + 1 ADD)

- **On superscalar RISC processors, out-of order (OOO) execution hardware is available to optimize the usage of the parallel hardware**

Courtesy of Prof. G. Wellein, U. of Erlangen

Superscalar Processors – Instruction Level Parallelism

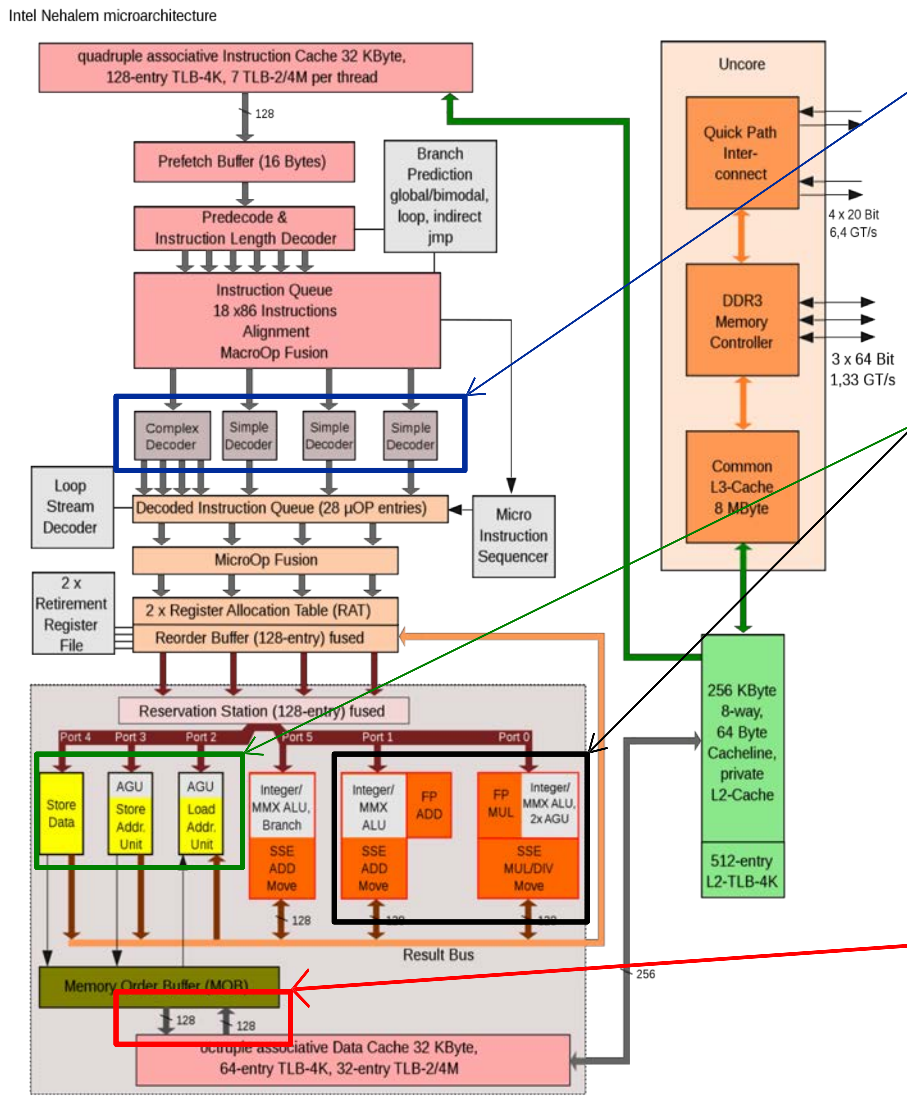
- Multiple units enable use of Instrucion Level Parallelism (ILP): Instruction stream is "parallelized" on the fly



- Issuing m concurrent instructions per cycle: m-way superscalar
- Modern processors (cores) are 3- to 6-way superscalar & can perform 2 or 4 floating point operations per cycle

Courtesy of Prof. G. Wellein, U. of Erlangen

Superscalar processor – Intel Nehalem design



Intel Nehalem microarchitecture
quadruple associative Instruction Cache 32 KByte, 128-entry TLB-4K, 7 TLB-2/4M per thread
128
Prefetch Buffer (16 Bytes)
Branch Prediction global/bimodal, loop, indirect jmp
Predecode & Instruction Length Decoder
Instruction Queue 18 x86 Instructions Alignment MacroOp Fusion
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
Loop Stream Decoder
Decoded Instruction Queue (28 µOP entries)
Micro Instruction Sequencer
MicroOp Fusion
2 x Retirement Register File
2 x Register Allocation Table (RAT)
Reorder Buffer (128-entry) fused
Reservation Station (128-entry) fused
Port 4
Port 3
Port 2
Port 5
Port 1
Port 0
Store Data
AGU Store Addr. Unit
AGU Load Addr. Unit
Integer/ MMX ALU, Branch
SSE ADD Move
Integer/ MMX ALU
SSE ADD Move
FP ADD
FP MUL
Integer/ MMX ALU, 2x AGU
SSE MUL/DIV Move
128
Result Bus
Memory Order Buffer (MOB)
128
128
octuple associative Data Cache 32 KByte, 64-entry TLB-4K, 32-entry TLB-2/4M
256
Uncore
Quick Path Inter-connect
4 x 20 Bit 6,4 GT/s
DDR3 Memory Controller
3 x 64 Bit 1,33 GT/s
Common L3-Cache 8 MByte
256 KByte 8-way, 64 Byte Cacheline, private L2-Cache
512-entry L2-TLB-4K

GT/s: gigatransfers per second

- **Decode & issue a max. of 4 instructions per cycle: IPC=4**
  →Min. CPI=0.25 Cycles/Instruction
- **Parallel units:**
  - FP ADD & FP MULT (work in parallel)
  - LOAD + STORE (work in parallel)
- **Max. FP performance:**
  - **1 ADD + 1 MULT instruction/cycle**
- **Max. performance:**
  - **`A(i) = r0 + r1 * B(i)`**
- **½ of max. FP performance:**
  - **`A(i) = r1 * B(i)`**
- **1/3 of max. FP performance:**
  - **`A(i) = A(i) + B(i) * C(i)`**

Software pipelining

- **Example:**

Assumptions:

Instructions block or "stall" the pipeline if operands are not available;

Assume c is in a register

Fortran Code:

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

Latencies

load a[i]	Load operand to register (4 cycles)
mult a[i] = c*a[i]	Multiply a(i) by c (2 cycles); a[i],c in registers
store a[i]	Write back result from register to mem./cache (2 cycles)
branch.loop	Increase loop counter if i ≤ N (0 cycles)

Simple Pseudo Code:

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

Lots of latency

Software pipelining

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

Naive instruction issue

Cycle 1	*load a[1]*
Cycle 2	
Cycle 3	
Cycle 4	
Cycle 5	*mult a[1]=c*a[1]*
Cycle 6	
Cycle 7	*store a[1]*
Cycle 8	
Cycle 9	*load a[2]*
Cycle 10	
Cycle 11	
Cycle 12	
Cycle 13	*mult a[2]=c*a[2]*
Cycle 14	
Cycle 15	*store a[2]*
Cycle 16	
Cycle 17	*load a[3]*
Cycle 18	
Cycle 19	

T= 96 cycles

Courtesy of Prof. G. Wellein, U. of Erlangen

Software pipelining

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

Naive instruction issue		Optimized instruction issue			
Cycle 1	*load a[1]*	*load a[1]*			Prolog (wind-up)
Cycle 2		*load a[2]*			
Cycle 3		*load a[3]*			
Cycle 4		*load a[4]*			
Cycle 5	*mult a[1]=c*a[1]*	*load a[5]*	*mult a[1]=c*a[1]*		
Cycle 6		*load a[6]*	*mult a[2]=c*a[2]*		
Cycle 7	*store a[1]*	*load a[7]*	*mult a[3]=c*a[3]*	*store a[1]*	**Kernel**
Cycle 8		*load a[8]*	*mult a[4]=c*a[4]*	*store a[2]*	
Cycle 9	*load a[2]*	*load a[9]*	*mult a[5]=c*a[5]*	*store a[3]*	
Cycle 10		*load a[10]*	*mult a[6]=c*a[6]*	*store a[4]*	
Cycle 11		*load a[11]*	*mult a[7]=c*a[7]*	*store a[5]*	
Cycle 12		*load a[12]*	*mult a[8]=c*a[8]*	*store a[6]*	
Cycle 13	*mult a[2]=c*a[2]*		*mult a[9]=c*a[9]*	*store a[7]*	Epilog (wind-down)
Cycle 14			*mult a[10]=c*a[10]*	*store a[8]*	
Cycle 15	*store a[2]*		*mult a[11]=c*a[11]*	*store a[9]*	
Cycle 16			*mult a[12]=c*a[12]*	*store a[10]*	
Cycle 17	*load a[3]*			*store a[11]*	
Cycle 18				*store a[12]*	
Cycle 19					

T= 96 cycles

T= 19 cycles

Courtesy of Prof. G. Wellein, U. of Erlangen

Software pipelining

- **Example:**

Assumptions:

Instructions block or "stall" the pipeline if operands are not available;

Assume c is in a register

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

load a[i]	Load operand to register (4 cycles)
mult a[i] = c*a[i]	Multiply a(i) by c (2 cycles); a[i],c in registers
store a[i]	Write back result from register to mem./cache (2 cycles)
branch.loop	Increase loop counter if i ≤ N (0 cycles)

Latencies

Simple Pseudo Code:
```
loop:     load a[i]
          mult a[i] = c*a[i]
          store a[i]
          branch.loop
```

Lots of latency

Optimized Pseudo Code:
```
loop:     load a[i+6]
          mult a[i+2] = c*a[i+2]
          store a[i]
          branch.loop
```

No latency

Courtesy of Prof. G. Wellein, U. of Erlangen

Efficient use of Pipelining

- **Software pipelining can be done by the compiler, but efficient reordering of the instructions requires deep insight into application (data dependencies) and processor (latencies of functional units)**

- **Re-ordering of instructions can also be done at runtime by the hardware: out-of-order (OOO) execution**

- **(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) * c
end do
```

Dependency:

```
do i=2,N
   a(i) = a(i-1) * c
end do
```

Pseudo-Dependency:

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

Pipelining: Beyond multiplication

- **Typical number of pipeline stages on modern CPUs:**
  - 2-5 for most (important) hardware pipelines: **L**oa**D**; **ST**ore; **MULT**; **ADD**
  - >>10 for other floating point pipelines: **DIV**ide/**SQ**uare**R**oo**T**

- **Most x86 processors (AMD, Intel):**
  1 MULT & 1 ADD floating point unit per processor core

- **Latest Intel (Haswell+):**
  2 Floating Point Fused MultiplyAdd (FMA) units
  - FMA3 instruction: s=s+a*b → 1 Input register (s) is overwritten
  - FMA4 instruction: s=r+a*b → No input register is modified

- **"Costs" of pipelined instructions**
  - Latency [cycles/instruction]: Depth of pipeline, i.e. cycles to execute a single instruction (worst case)
  - Throughput [instructions/cycle or results/cycle]: results per cycle for filled pipeline (best case = 1 result/ cycle for scalar operations with 1 arithmetic unit)

Courtesy of Prof. G. Wellein, U. of Erlangen

Expensive instructions

- Examples for Intel Sandy Bridge processors

	Operation	Instruction Latency [cy]	Throughput: Scalar Cycles per Result [cy]	Throughput: AVX Cycles per Result [cy]
instructions	ADD (DP/SP)	3 / 3	1 / 1	0.25 / 0.125
	MULT (DP/SP)	5 / 5	1 / 1	0.25 / 0.125
	SQRT (DP)	**45**	**44**	**11**
	SQRT (SP)	**29**	**28**	**7** SIMD
	DIV (DP)	**45**	**44**	**11**
Library calls	EXP (DP)	**83 (glibc 2.12)**	**83 (glibc 2.12)**	**12.5 (SVML)**
	SIN (DP)	**79 (glibc 2.12)**	**79 (glibc 2.12)**	**17.5 (SVML)**

- Lesson: Avoid expensive instructions!

Pipelining: Potential problems (1)

- Hidden data dependencies:

```
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 "Pointer Aliasing" , i.e. `A` → `&C[-1]` ; `B` → `&C[-2]` → `C[i] = C[i-1] + C[i-2]` → Dependency!

- Compiler cannot resolve potential pointer aliasing conflicts on its own!

- If no "Pointer Aliasing" is used, be sure to tell the compiler, e.g.
  - use `-fno-alias` switch for Intel compiler
  - Pass arguments as `(double *restrict A,…)` (only as of C99 standard)

Pipelining: Potential problems (2)

- Simple subroutine/function calls within a loop

```
do i=1, N
    call elementsum(A(i),B(i), psum)
    C(i)=psum
enddo
…
function elementsum( a, b, psum)
…
psum=a+b
```

→ Inline subroutines (can be done by compiler….)

```
do i=1, N
    psum=A(i)+B(i)
    C(i)=psum
enddo
…
```

Pipelining: Potential problems (3)

- **What about “reduction operations”?**

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

→

`A(i)` (incl. LD)

`sum` in register xmm1

```
..B1.62:                                    # Preds ..B1.62 ..B1.61
        addsd     test_(,%rdx,8), %xmm1
        incq      %rdx
        cmpq      %rax, %rdx
        jb        ..B1.62                   # Prob 99%
```

ADD 1st argument to 2nd argument and store result in 2nd argument

`N`

`i` (loop counter)

- **Benchmark: Run above assembly language kernel with N=32,64,128,…,4096 on processor with**
  - 3.5 GHz clock speed
  - 1 pipelined ADD unit (latency 3 cycles)
  - 1 pipelined LOAD unit (latency 4 cycles)

→ Clk Spd=3500 Mcycle/s

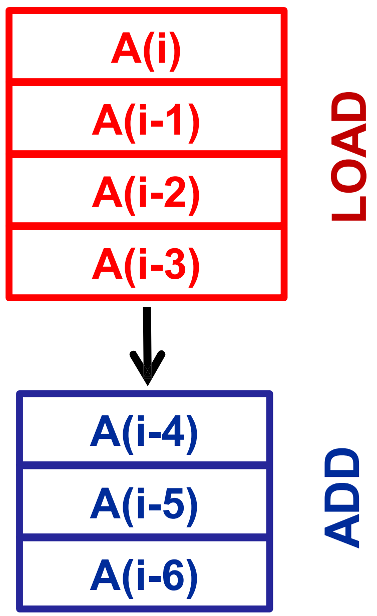
→ **1 iteration per cycle**
(after 7 iterations)

Pipelining: Potential problems (4)

- Expected Performance: Throughput * ClockSpeed

- Throughput: N/T(N) = N/ (L+N)
  Assumption: L is total latency of one iteration. One result is delivered each cycle after pipeline startup.
  → Total runtime: (L+N) cycles

- Total latency: L = 4 cycles + 3 cycles = 7 cycles



→ Performance for N Iterations:

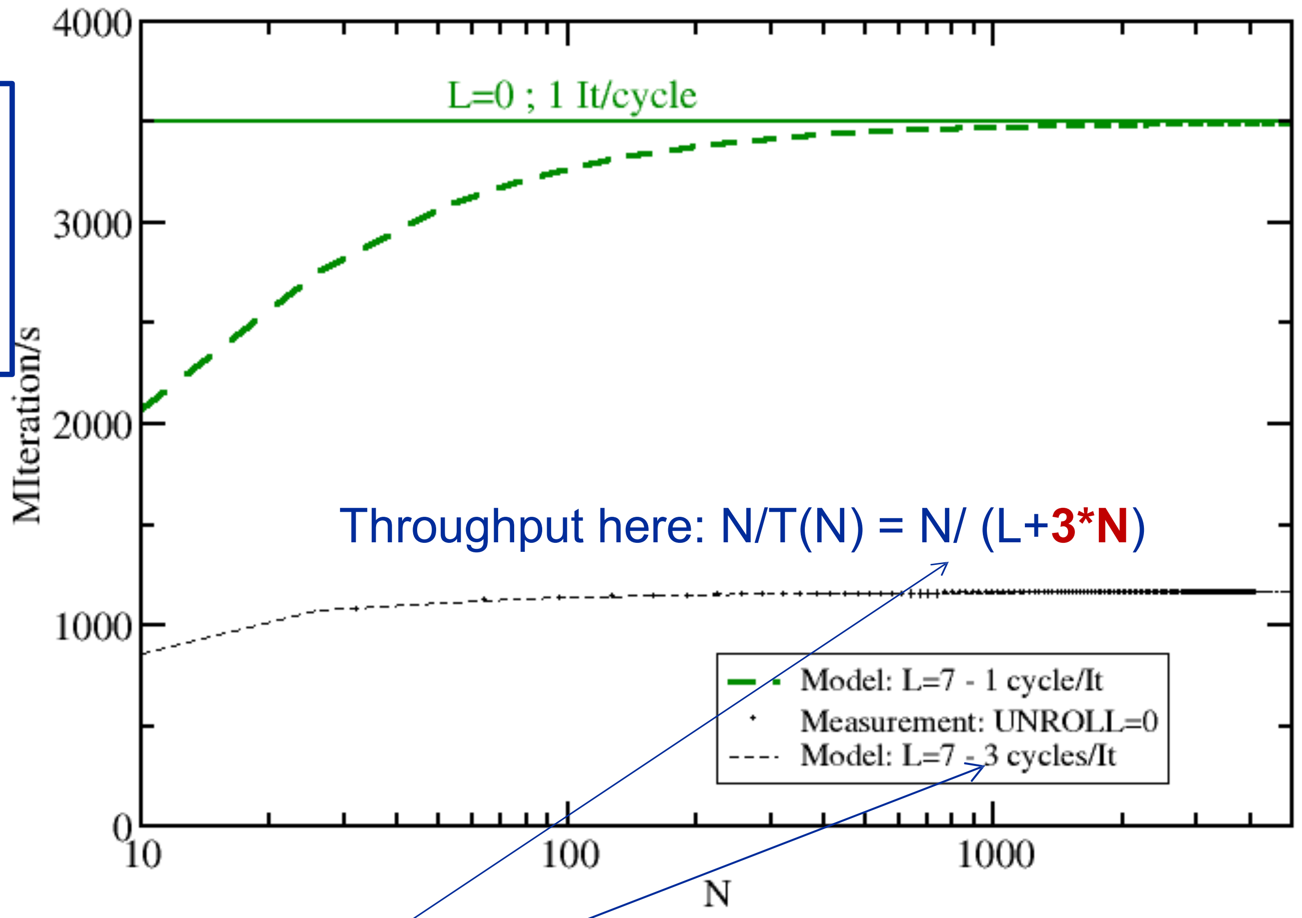
3500 MHz * (N / (L+N)) Iterations/cycle

→ Maximum performance ($N \to \infty$):

3500 Mcycle/s * 1 Iteration/cycle=
3500 Miterations/s

Pipelining: Potential problems (5)

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



Throughput here: N/T(N) = N/ (L+**3*N**)

Dependency on `sum` → next instruction needs to wait for completion of previous one → only 1 out of 3 stages active → 3 cycles per iteration

Pipelining: Potential problems (6)

- **Increase pipeline utilization by "loop unrolling"**

"2-way Unrolling" (N is even)

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

$\downarrow$

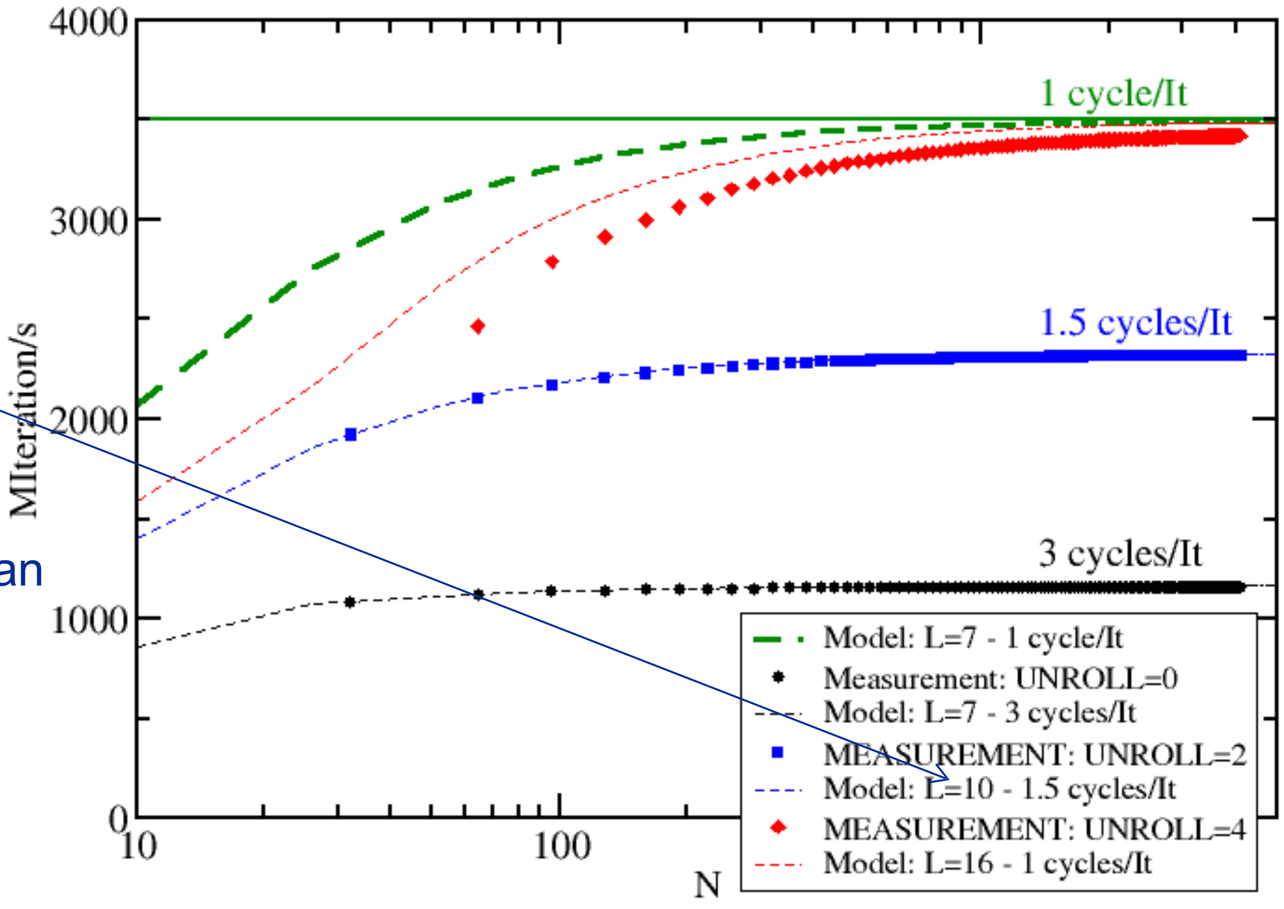
2 out of 3 pipeline stages can be filled

$\downarrow$

2 results every 3 cycles

$\downarrow$

1.5 cycle/Iteration



Courtesy of Prof. G. Wellein, U. of Erlangen

Pipelining: Potential problems (7)

- 4-way Unrolling (actually at least 3-way) to get best performance

- Sum is split up in 4 independent partial sums

- Compiler can do that, if it is allowed to do so…
- Computer's floating point arithmetic is not associative!

$$\left(\left(\left((a+b)+c\right)+d\right)+e\right)+f \neq (a+b)+(c+d)+(e+f)$$

- If you require binary exact result (`-fp-model strict`) compiler is not allowed to do this transformation
- L=(7+3*3) cycles (see prev. slide) (Best case with no remainder loop)

"4-way Unrolling"

```
Nr=4*(N/4)
sum1=0.d0
sum2=0.d0
sum3=0.d0
sum4=0.d0
do i=1, Nr, 4
   sum1=sum1+A(i)
   sum2=sum2+A(i+1)
   sum3=sum3+A(i+2)
   sum4=sum4+A(i+3)
enddo
do i=Nr+1, N
   sum1=sum1+A(i)
enddo
sum=sum1+sum2+sum3+sum4
```

Remainder loop

Courtesy of Prof. G. Wellein, U. of Erlangen