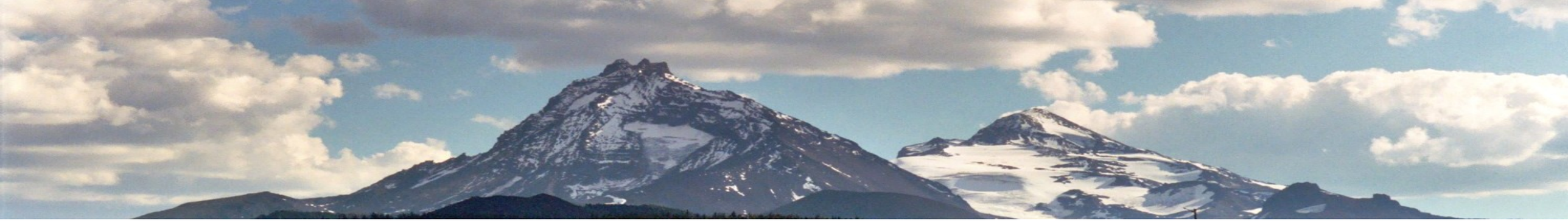


[A]rchitecture et [O]ptimisation de [C]ode pour microprocesseur hautes performances

Processor Architecture

Presented by

Emmanuel Oseret, Salah Ibnamar
Exascale Computing Research
University of Versailles St Quentin en Yvelines

William Jalby
University of Versailles St Quentin en Yvelines

Original version by *Allen D. Malony*, department of Computer and Information Science (University of Oregon), modified by *Emmanuel Oseret*.

Course Plan

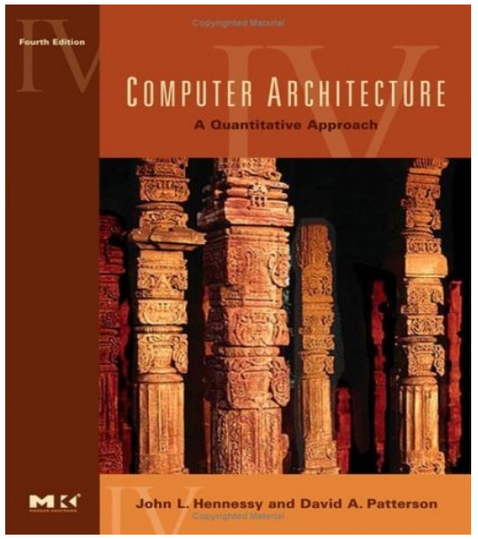
- Evolution of computer architectures
  - Scalar
  - Superscalar
  - Vector
- Hierarchical memory systems
  - Caches
  - Translation Lookaside Buffer (TLB)
- Code optimizations for CPU (micro) architectures
  - Scheduling
  - Loop transformations
- Multicore architectures

Origin of Course Material

- J. Fisher, HP Labs (Palo Alto, US)
- D. Patterson, UC Berkeley (US)
- D. Culler, UC Berkeley (US)
- S. Niar, Université de Valenciennes (FR)
- J.N. Amaral, University of Alberta (CA)
- R. Gupta, University of Arizona (US)
- S. Mahlke, University of Michigan (US)
- D. Etiemble, Université de Paris-Sud (FR)
- O. Sentieys, IRISA (FR)
- S. Touati, Sophia Antipolis (FR)

Origin of Course Materials

- J. Hennessy and D. Patterson, Computer Architecture: A Quantitative Approach, 5th Edition, ISBN: 9780123838728, Morgan Kaufmann, 2011.



- Wikipedia

WIKIPEDIA
The Free Encyclopedia

Introduction to CPU Architectures

- Scalar processors
  - Basic architecture
  - Pipeline (within an instruction)
  - Hazards and solutions
    - Data (out of order execution)
    - Control (branch prediction)
- Executing multiple instructions simultaneously
  - Pipelining between instructions
  - Overlapped execution
  - Multiple functional units
  - Out of order execution
  - Multi-issue execution
  - Superscalar and superpipelining
  - Very Long Instruction Word (VLIW)
  - Hardware multithreading (hyperthreading)
- SIMD and vector processing

Quantitative Principles

- Computers operate at a particular clock rate in discrete time (clock) events: *ticks*, *periods*, *cycles*
- Calculate the total time (T) to execute a program

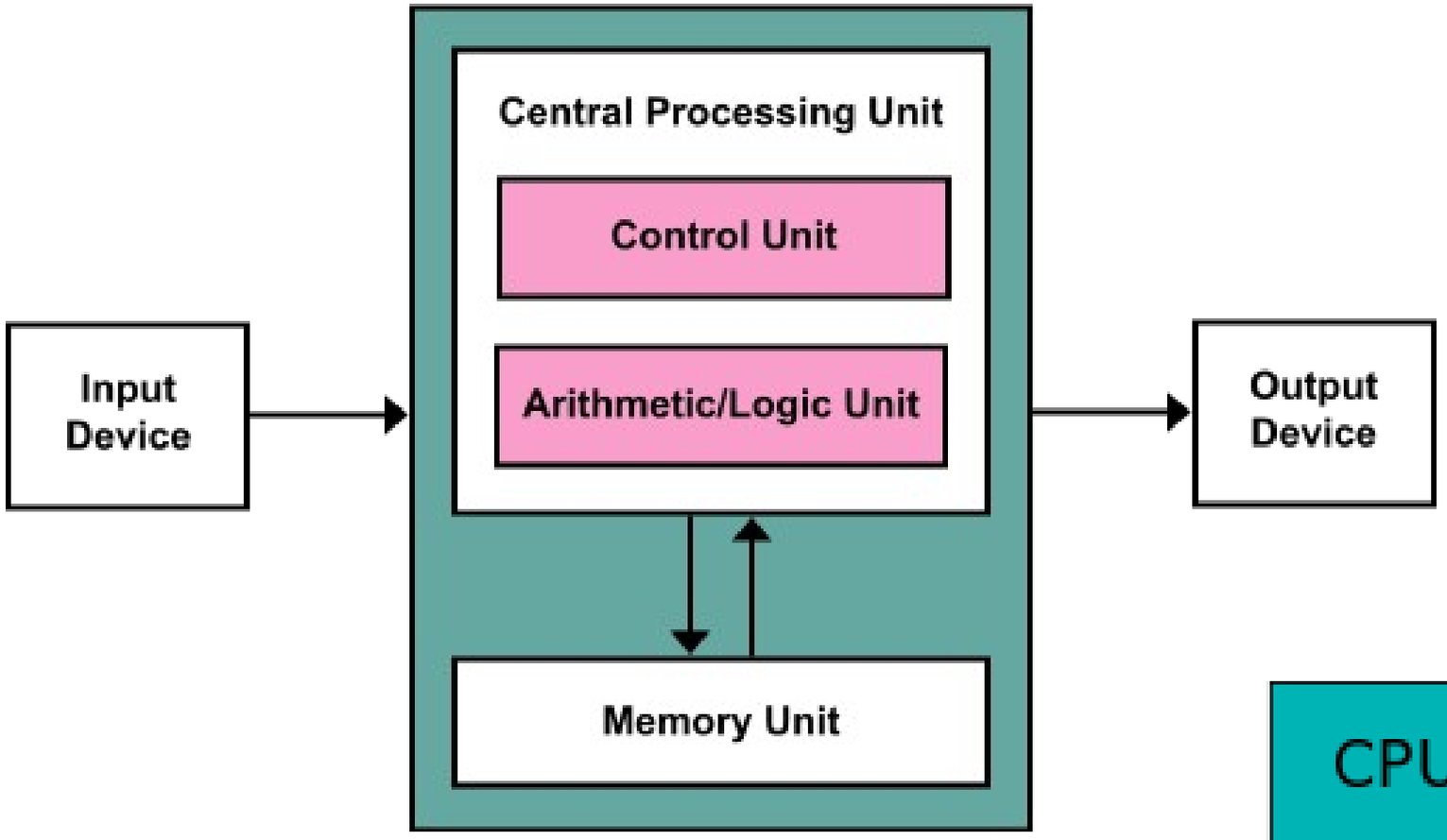
  T = (total clock cycles for program) / (clock rate)
- Let *CPI* be the # of cycles to execute an instruction
- Let *IC* be the total # instructions executed
- CPU performance equation (total execution time)

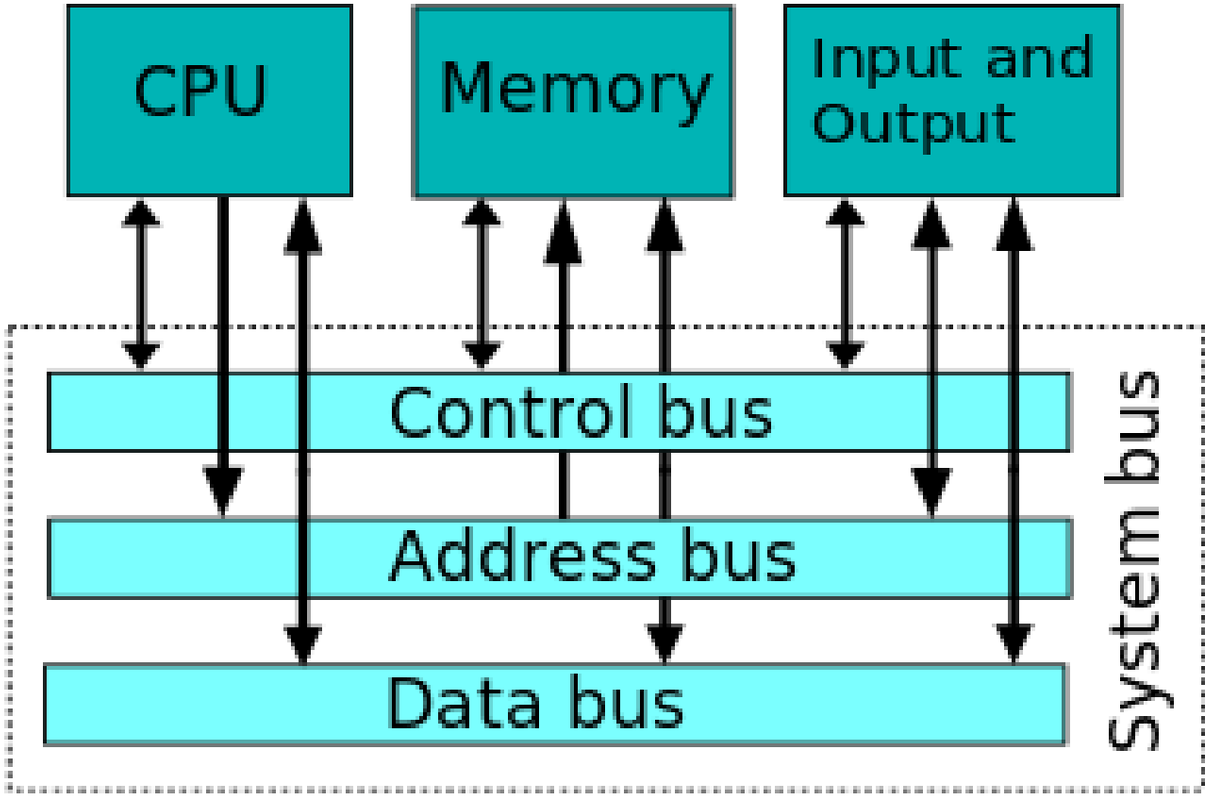
  *T = (IC * CPI) / (clock rate)*

"Iron Law" of CPU Performance

- CPU performance is dependent on 3 parameters:
  - *Clock cycle* (rate): hardware, organization
  - *Instruction count*: ISA, compiler
  - *Clock cycles per instruction*: ISA, organization
- These are not mutually exclusive
- Different instructions take different # cycles
- CPI is harder to determine because CPU architecture optimizations try to optimize instruction execution
- Instructions per cycle (*IPC = 1 / CPI*)
  - Objective is to increase the IPC

Von Neumann Computer Architecture





Source : Wikimedia

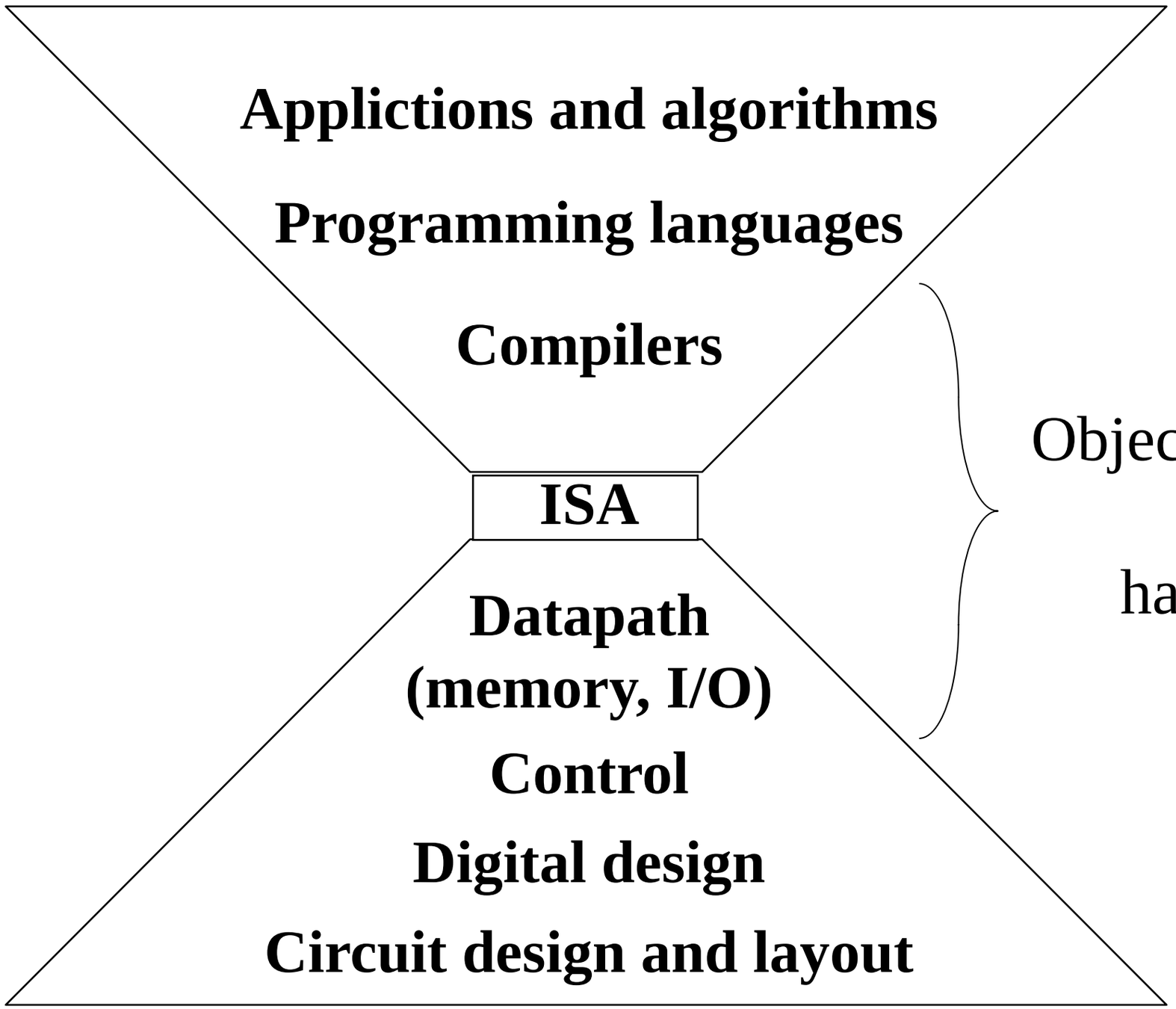
Computer Architecture

- Architecture
  - What is visible to the programmer
- Components
  - Central processing unit (CPU)
  - Instruction set architecture
    - logical and arithmetic operations
  - Memory model
    - CPU data registers
    - memory
  - Memory addressing modes
    - register-register
    - memory-register
    - memory-memory

Architecture versus Micro-architecture

- Processor architecture is concerned with what is visible to the programmer
- Micro-architecture defines:
  - Internal processor structures and mechanisms of a processor necessary to implement the ISA
  - What happens when an instruction is executed

Architecture Levels



Objective of the course is to look at the hardware/software interface

Source : David Patterson

Performance Improvement

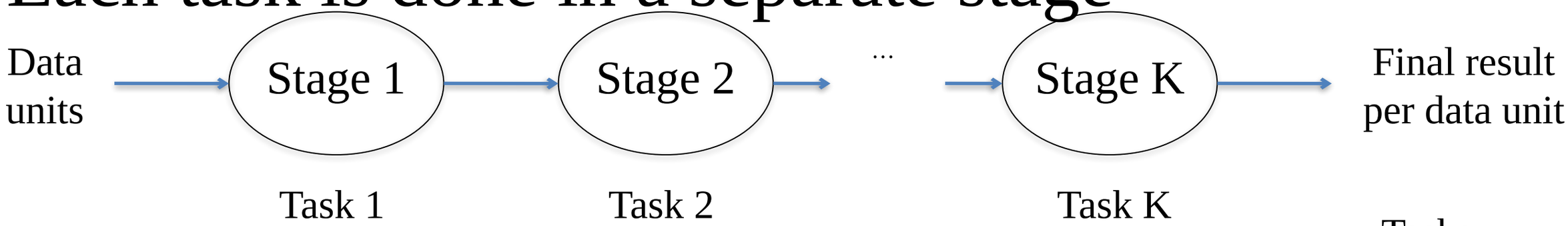
- For a particular architecture, how do we improve performance?
  - First idea might be to just increase the clock frequency
  - Improve the efficiency of components
    - reduces the time needed to perform the operations
  - Reduce the cost of certain operations in the processor
  - Must be concerned with the physical limits
    - IC technology used (circuit line width)
    - power consumption and heat dissipation
-

Pipeline

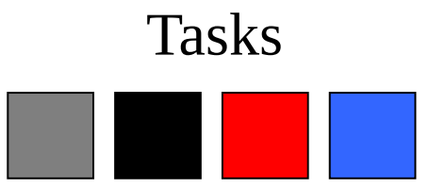
- A pipeline is a linear sequence of stages
- Data flows through the pipeline
  - From stage 1 to the last stage
  - Each stage performs some task
    - uses the result from the previous stage
  - Data is thought of as being composed of units (items)
  - Each data unit can be processed separately in pipeline
  - Sequence of stages (tasks) matters (functional operation)
  - Data sequence might matter
- Pipeline computation is a special form of *producer-consumer* parallelism
  - Producer tasks output data …
  - … used as input by consumer tasks

Pipeline Model

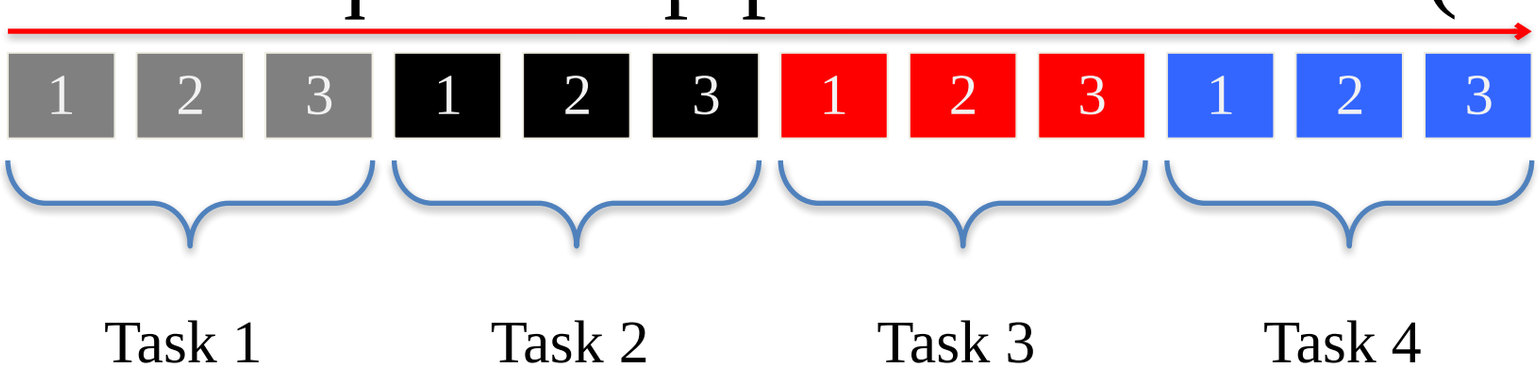
- *Stream* of data operated on by succession of tasks
  - Assumption: data input and output must be in sequence
- Each task is done in a separate stage

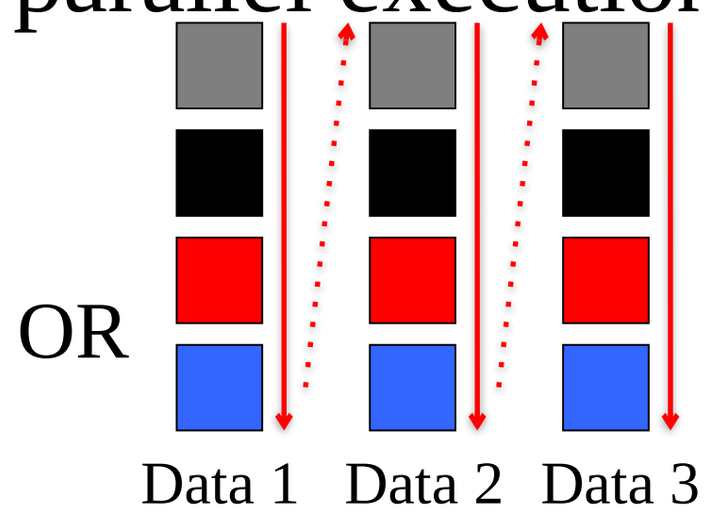


- Consider 3 data units and 4 tasks (stages)



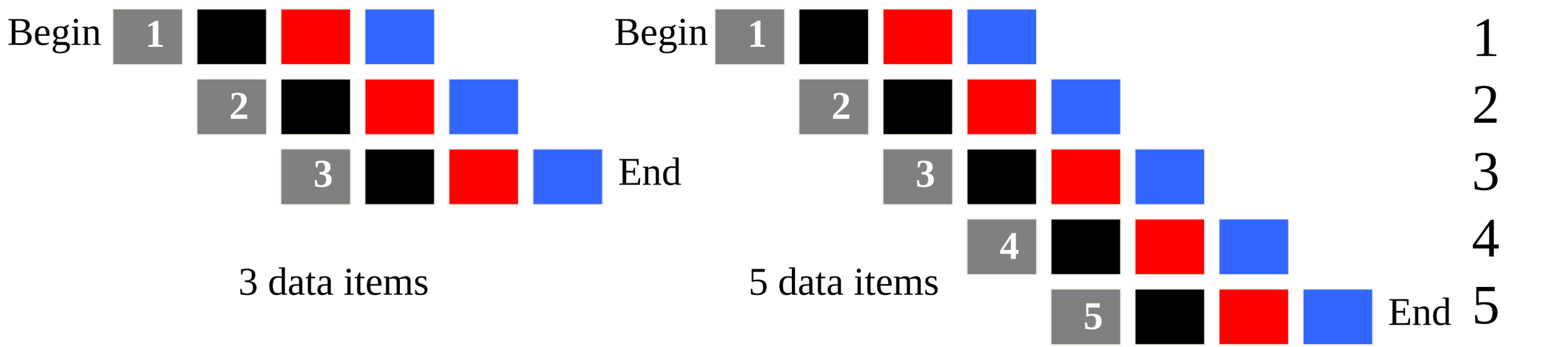
  - Sequential pipeline execution (no parallel execution)





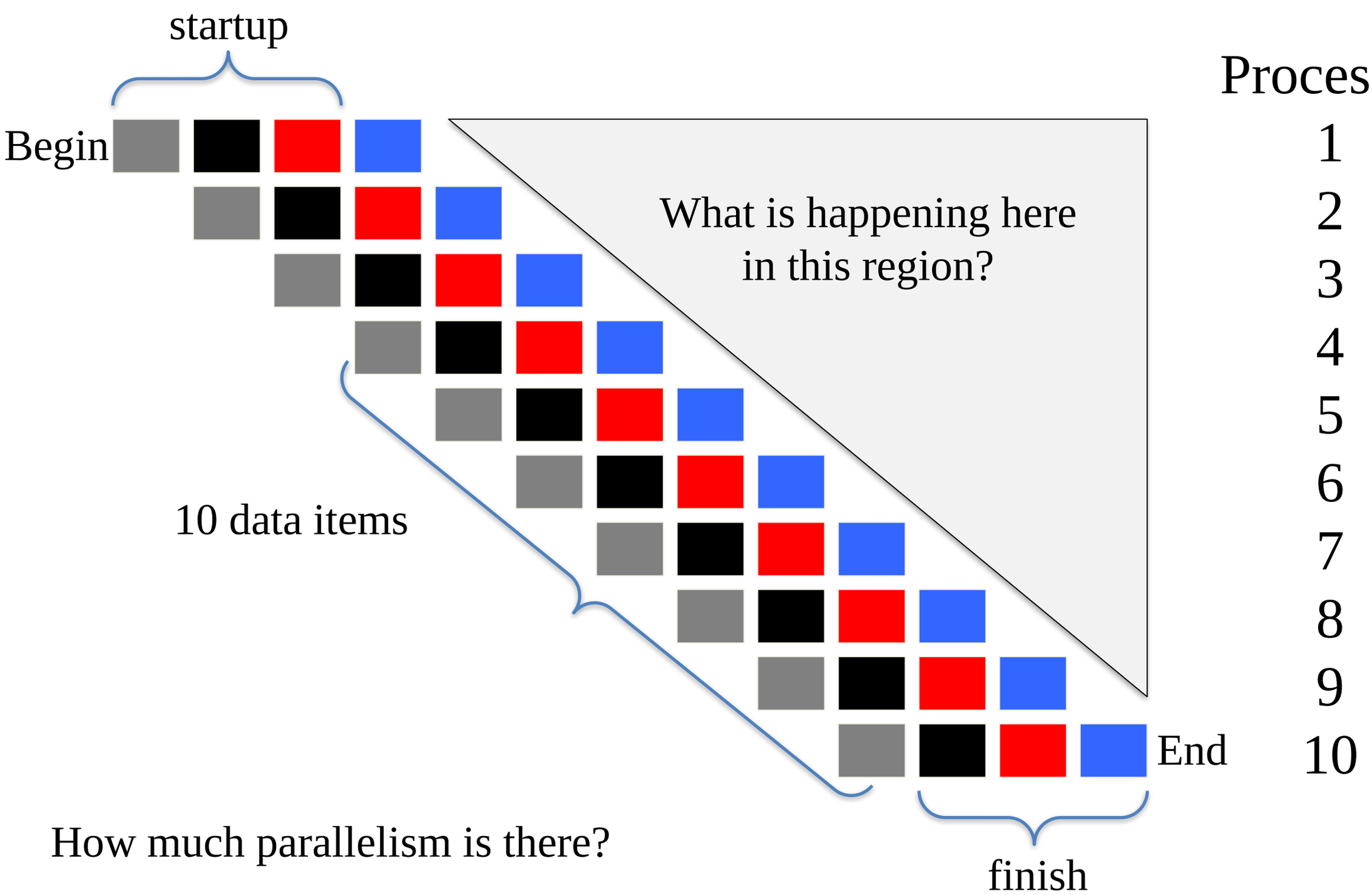
Where is the Concurrency? (Serial Pipeline)

- Pipeline with serial stages
  - Each stage runs serially (i.e., can not be done in parallel)
  - Assume that we can not parallelize the <u>tasks</u> (for now)
- What can we run in parallel?
  - Think about data parallelism
  - Provide a separate pipeline for each data item



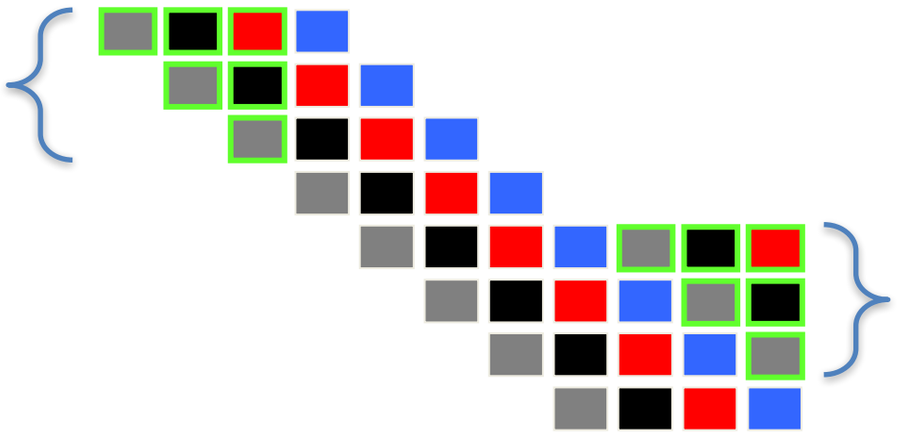
- What do you notice as we increase # data items?

Where is the Concurrency? (Serial Pipeline)



How much parallelism is there?

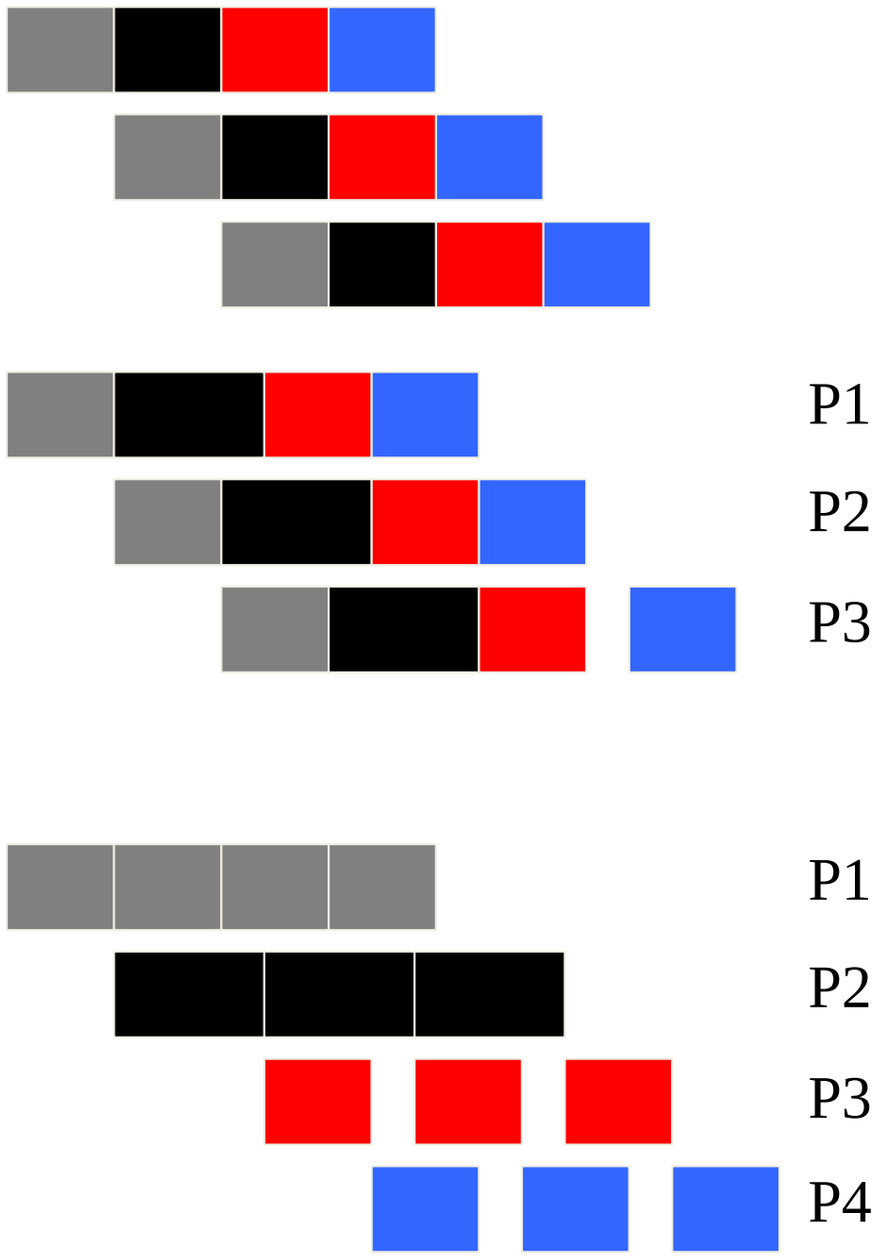
Pipeline Performance



- N data and T tasks
- Each task takes unit time t
- Sequential time = N*T*t = NT (t=1)
- *Parallel pipeline time*
  = start + finish + parallel
  = T-1 + T-1 + (N-2(T-1))/T
  = 2T-2 + N/T + (2T-2)/T = N/T + (2T-2)(1+1/T)
  = O(N/T) (for N>>T)
- Try to find a lot of data to pipeline
- Try to divide computation in a lot of pipeline tasks
  - More tasks to do (longer pipelines) = more parallelism
  - Shorter tasks to do (as a result of breaking apart tasks)
- Interested in pipeline throughput

Pipeline Performance

- *N* data and *T* tasks
- Suppose the tasks execution times are non-uniform
- Suppose a processor is assigned to execute a task
- What happens to the throughput?
- What limits performance?
- Slowest stage limits throughput … Why?
- Little's Law comes into play



Basic Throughput Quantities

- At all levels of the system (register files through networks), there are three fundamental (efficiency-oriented) quantities:
  - **Latency**
    - every operation requires time to execute
    - # stages (tasks) in a pipeline * time for each task
  - **Bandwidth**
    - # of (parallel) operations completed per cycle
  - **Concurrency**
    - total # of operations in flight

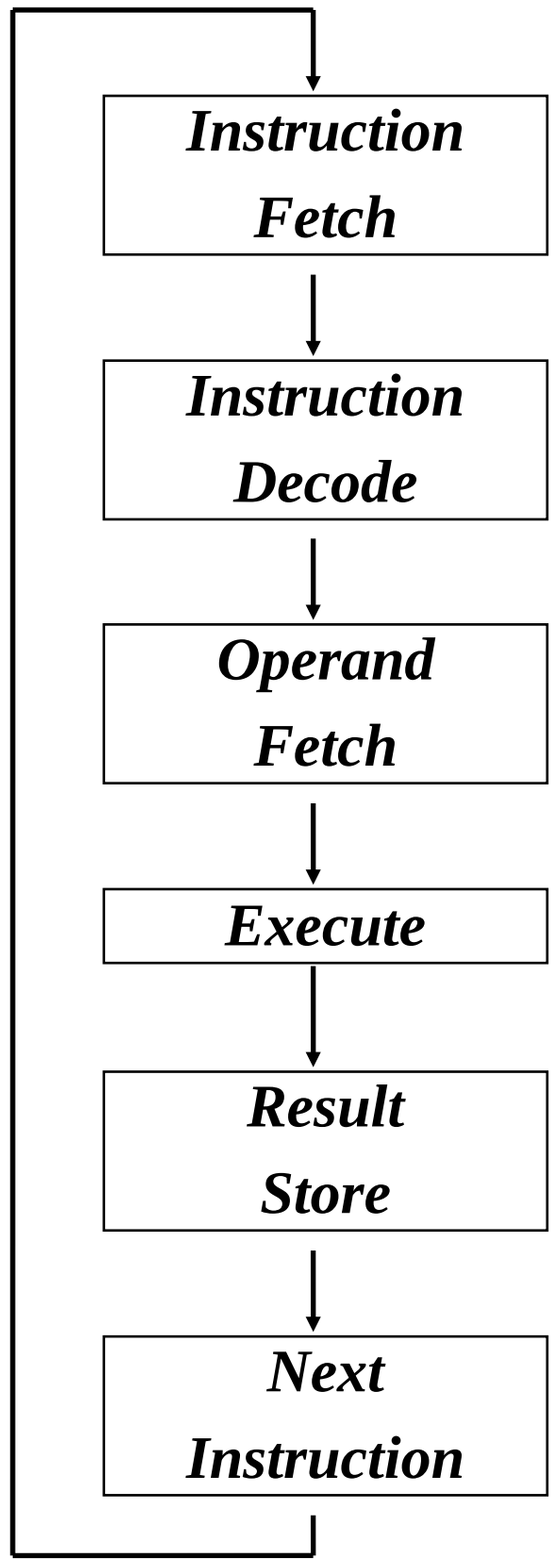
Little's Law

- In queueing theory, Little's Law expresses a relationship between latency and throughput in a stable system (e.g., a pipeline with infinite data)

  *occupancy (concurrency) = latency * throughput*

- Think of a water pipe
  - Throughput = rate at which water is put into the pipe
  - Latency = "length" of the pipe (processing time for data)
- If increase throughput, need more concurrency
  - How do you get it? … increase # pipeline stages
- If latency decreases, need more concurrency

Instruction Execution Cycle



Obtain instruction from program storage

Determine required actions and instruction size

Locate and obtain operand data

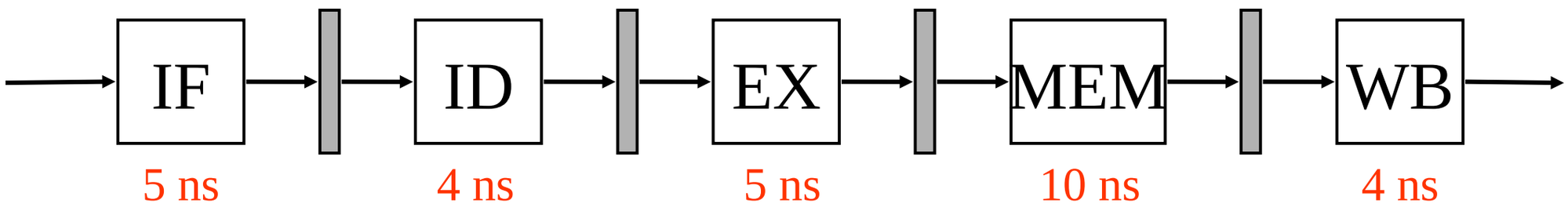
Compute result value or status

Deposit results in storage for later use

Determine successor instruction

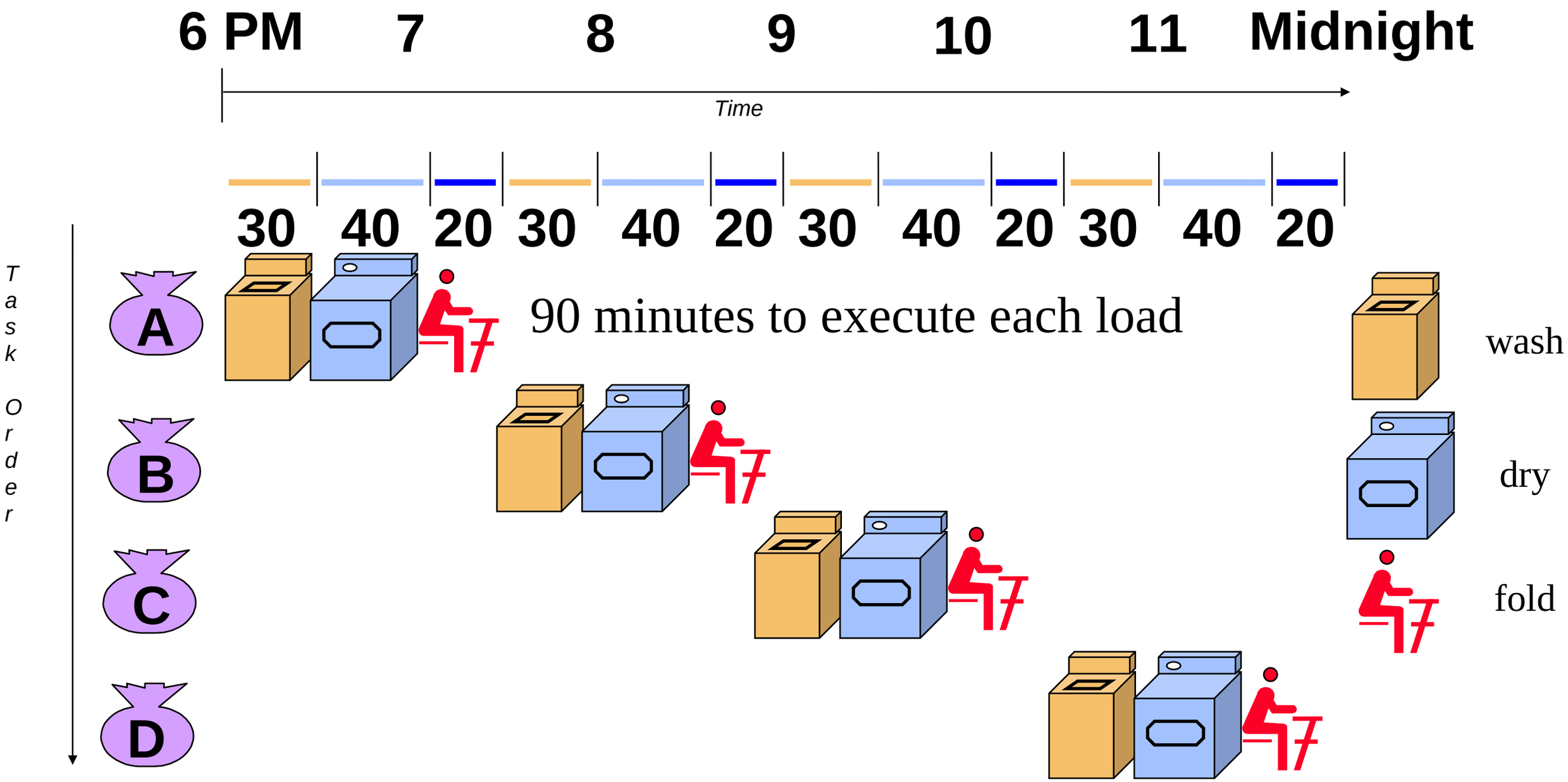
Instruction Execution Cycle (Stages)

- Consider steps for executing an instruction as stages:
  - **IF**: *Instruction Fetch* (load instruction from memory)
  - **ID**: *Instruction Decode*
  - **EX**: *Execution*
  - **MEM**: *Memory Access*
  - **WB**: *Write Back* (write result back to memory)



- Each instruction passes through all of these stages
- Different stages may take different time to execute

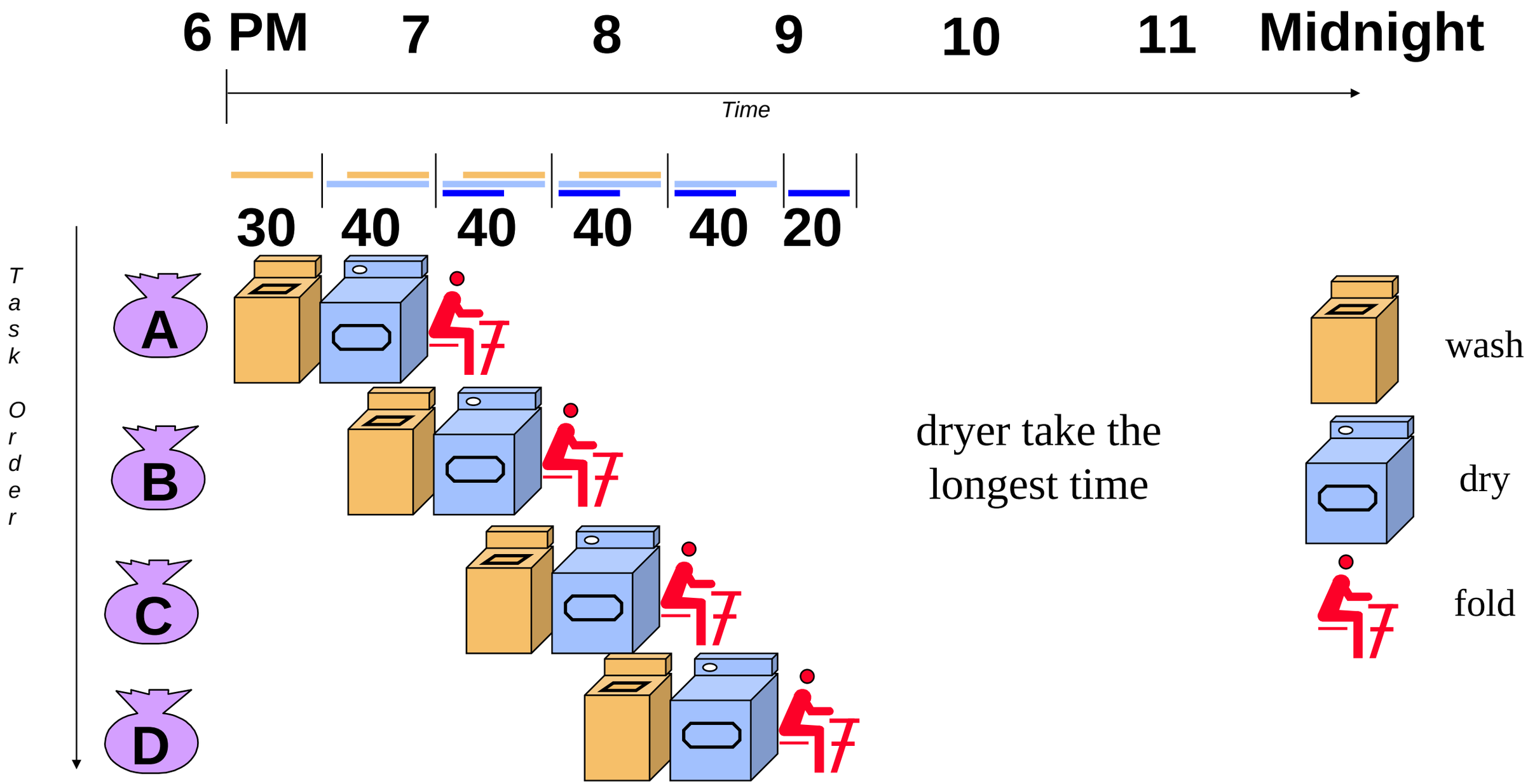
Consider a Sequential Laundry



- Sequential laundry takes 6 hours for 4 loads (jobs)
- How can we make this go faster?

Source : David Patterson

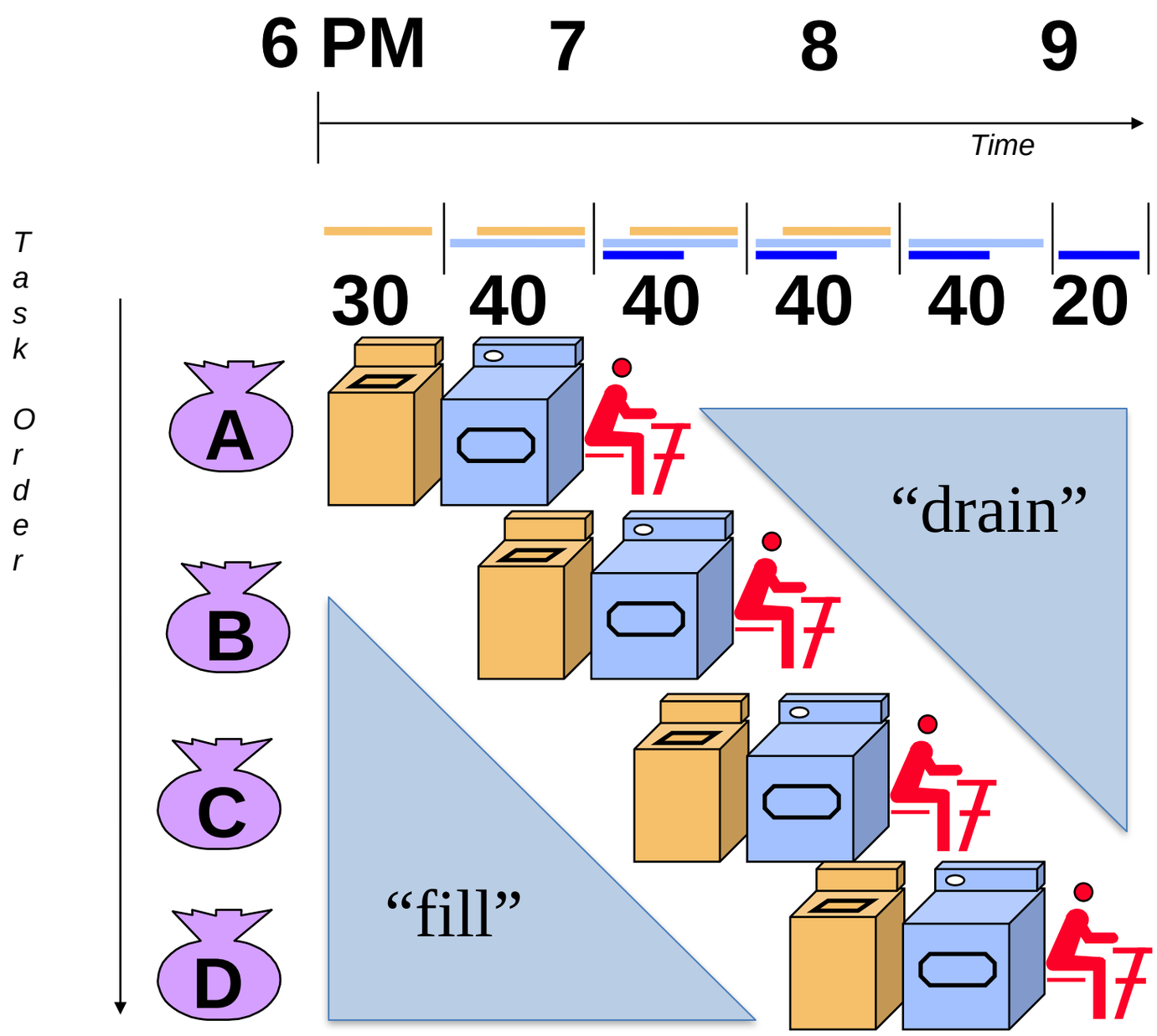
Laundry "Pipeline"



- Use resources as soon as they are available
- Pipelined laundry takes 3.5 hours for 4 loads

Source : David Patterson

Pipelining Lessons



- Pipelining does not help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" / "drain"

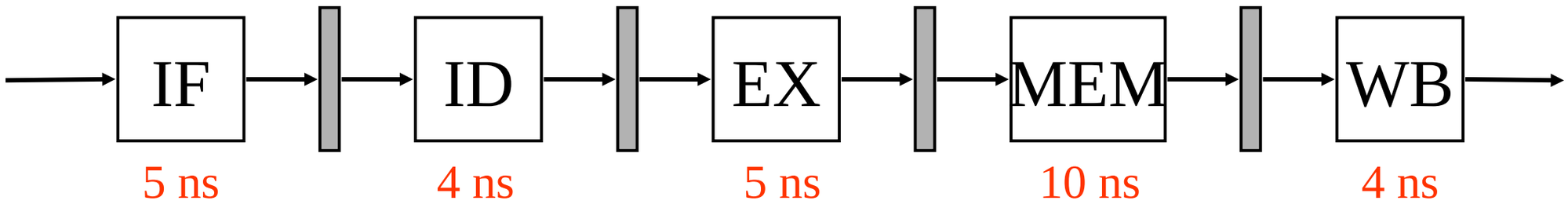
Pipeline Characteristics (for instructions)

- All instructions that are executed go through the same stages of the pipeline
- However, it is possible to overlap the execution of an instruction with other instructions
- *Latency* is the time it takes to execute a job
  - 90 minutes in the laundry case (wash + dry + fold)
- *Throughput* is the total number of jobs that can be completed in a certain amount of time

Source: J.N. Amaral

Sequential Pipeline Flow and Latency

- What is the latency of the pipeline?



L = *lat(IF)* + *lat(ID)* + *lat(EX)* + *lat(MEM)* + *lat(WB)*

= 5ns + 4ns + 5ns + 10ns + 4ns = 28ns

- Assume only 1 instruction is executing at a time
- How long does it take to execute the pipeline?

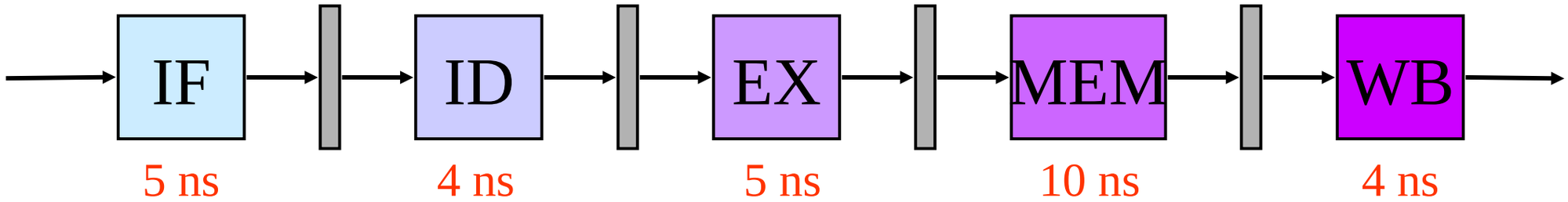
T = IC / L

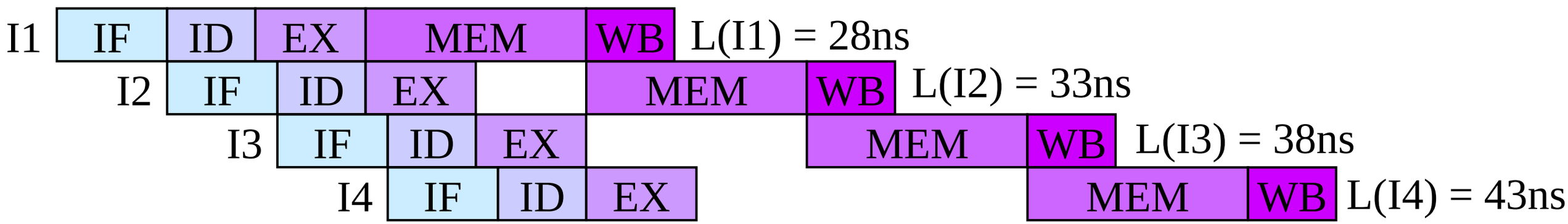
= IC / (5+4+5+10+4) ns

= IC / 28ns

Overlapping Instructions in the Pipeline

- Suppose we try to execute several instructions



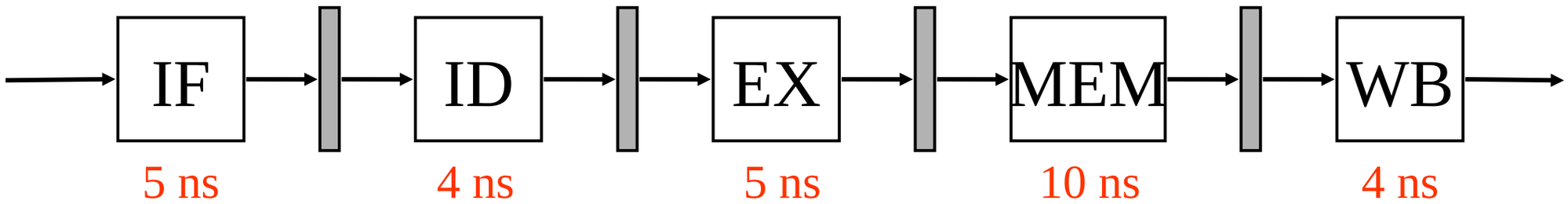


- Do you see any problems?
- Latencies are not constant
- Pipeline stages are not balanced

Source: J.N. Amaral

Pipeline Flow and Latency

- What is the latency of the (sequential) pipeline?



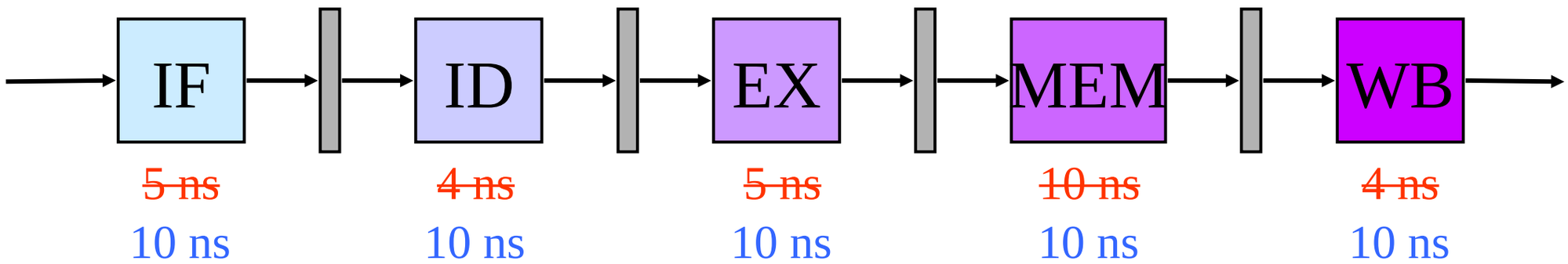
L = *lat(IF)* + *lat(ID)* + *lat(EX)* + *lat(MEM)* + *lat(WB)*
= 5ns + 4ns + 5ns + 10ns + 4ns = 28ns

- What if you overlap instructions?
- How long does it take to execute the pipeline?

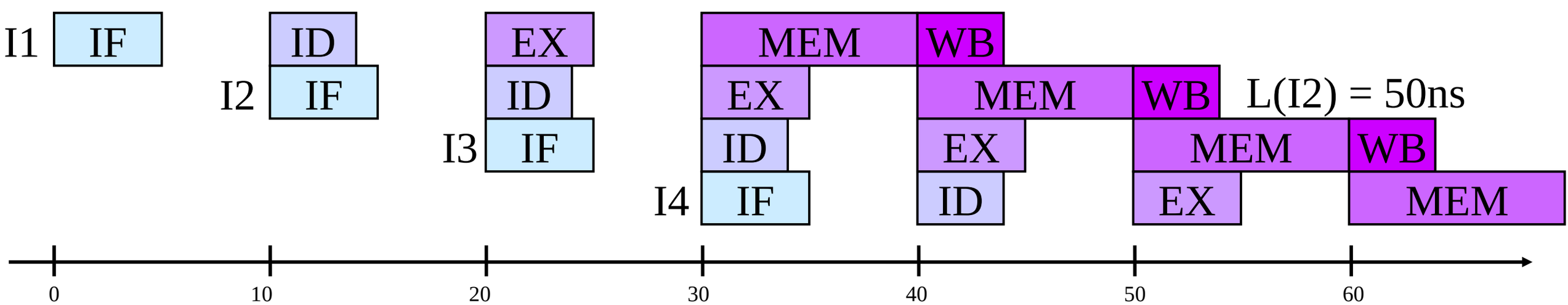
T = IC / max [*lat(IF),lat(ID),lat(EX),lat(MEM),lat(WB)*]
= IC / max [5,4,5,10,4]*ns*
= IC / 10*ns*

Balance the Instruction Pipeline Stages

- Make each instruction stage take the same time
  - Set to the longest stage



- Latencies are now the same ("balanced")

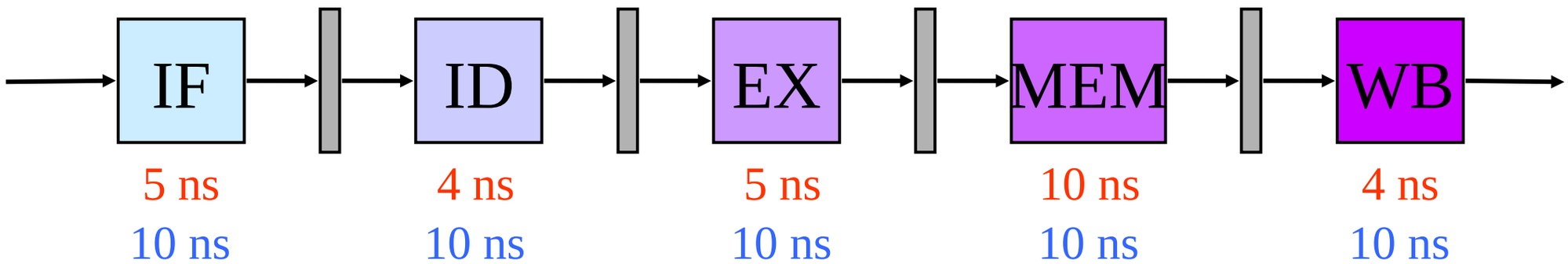


L(I1) = L(I2) = L(I3) = L(I4) = 50ns (> 28ns)

Source: J.N. Amaral

Computing CPU Time

- With a balanced pipeline, it is easier to calculate the CPU execution time for a program



- Suppose we execute 20000 instructions

$$T_{non-pipe} = 20000 \times 28ns = 560000ns = 560\mu s$$

$$T_{pipe} = 50 + 19999 \times 10ns = 200040ns = 200{,}04\mu s$$

Cost for startup and shutdown

Source: J.N. Amaral

Pipelining Speedup (Acceleration)

- Pipelining can deliver a performance improvement
- Measure by speedup:

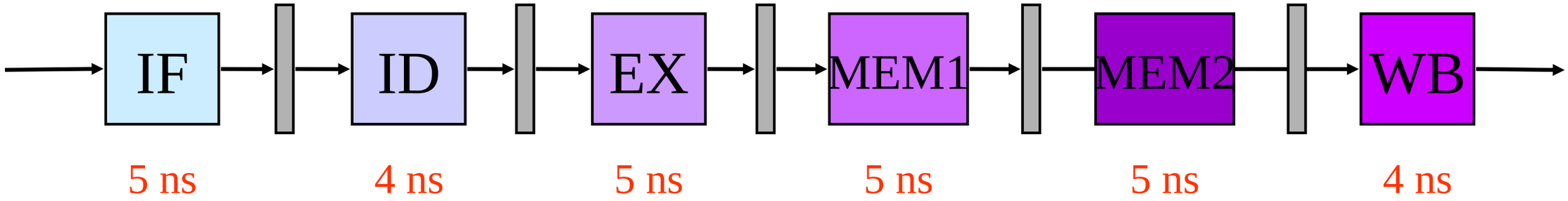
$$Speedup_{pipe} = \frac{ExecTime_{non\text{-}pipe}}{ExecTime_{pipe}} = \frac{560\mu s}{200{,}04\mu s} \approx 2.8$$

- Pipeline performance can be improved by:
  - Increasing the # stages
  - Decreasing the time of the slowest stage
  - Reducing the imbalance between stages

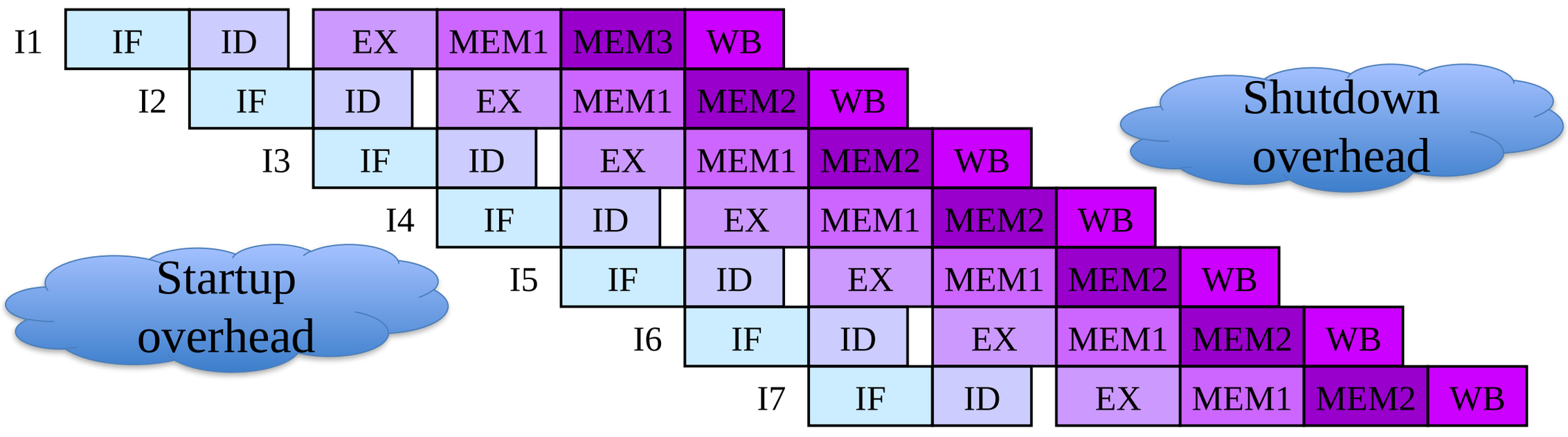
Source: J.N. Amaral

Split a Stage to Reduce Maximum Delay

- Suppose we split the memory access stage



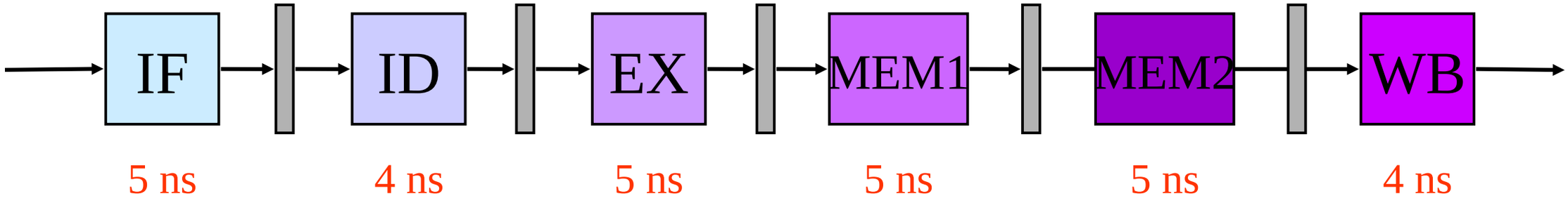
- Sequential latency is the same
- Pipeline execution of multiple instructions



Source: J.N. Amaral

Split a Stage to Reduce Maximum Delay

- Suppose we split the memory access stage



- Sequential latency is the same
- Pipeline (balanced) latency is reduced

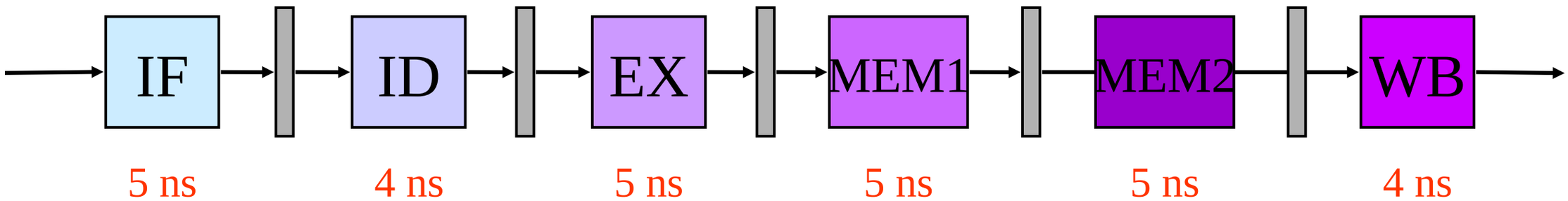
$$L = 6 \times 5ns = 30ns$$

- Total execution time is smaller (by 2 times!)

T = IC / max [*lat(IF),lat(ID),lat(EX),lat(M1),lat(M2),lat(WB)*]
= IC / max [5,4,5,5,5,4]*ns*
= IC / 5*ns*

Recalcuate the Pipeline Speedup

- Now pipeline stages have less delay variance



- We can recalculate the total time and speedup

$$ExecTime_{pipe} = 30 + 19999 \times 5ns = 100000ns = 100{,}25\mu s$$

$$Speedup_{pipe} = \frac{ExecTime_{non\text{-}pipe}}{ExecTime_{pipe}} = \frac{560\mu s}{100{,}025\mu s} \approx 5.6$$

- Notice that the startup and shutdown overhead improved because of the smaller maximum stage

Source: J.N. Amaral

Theoretical Performance

- Pipeline speedup is determined by
  - # stages (N)
  - Maximum stage delay (D)

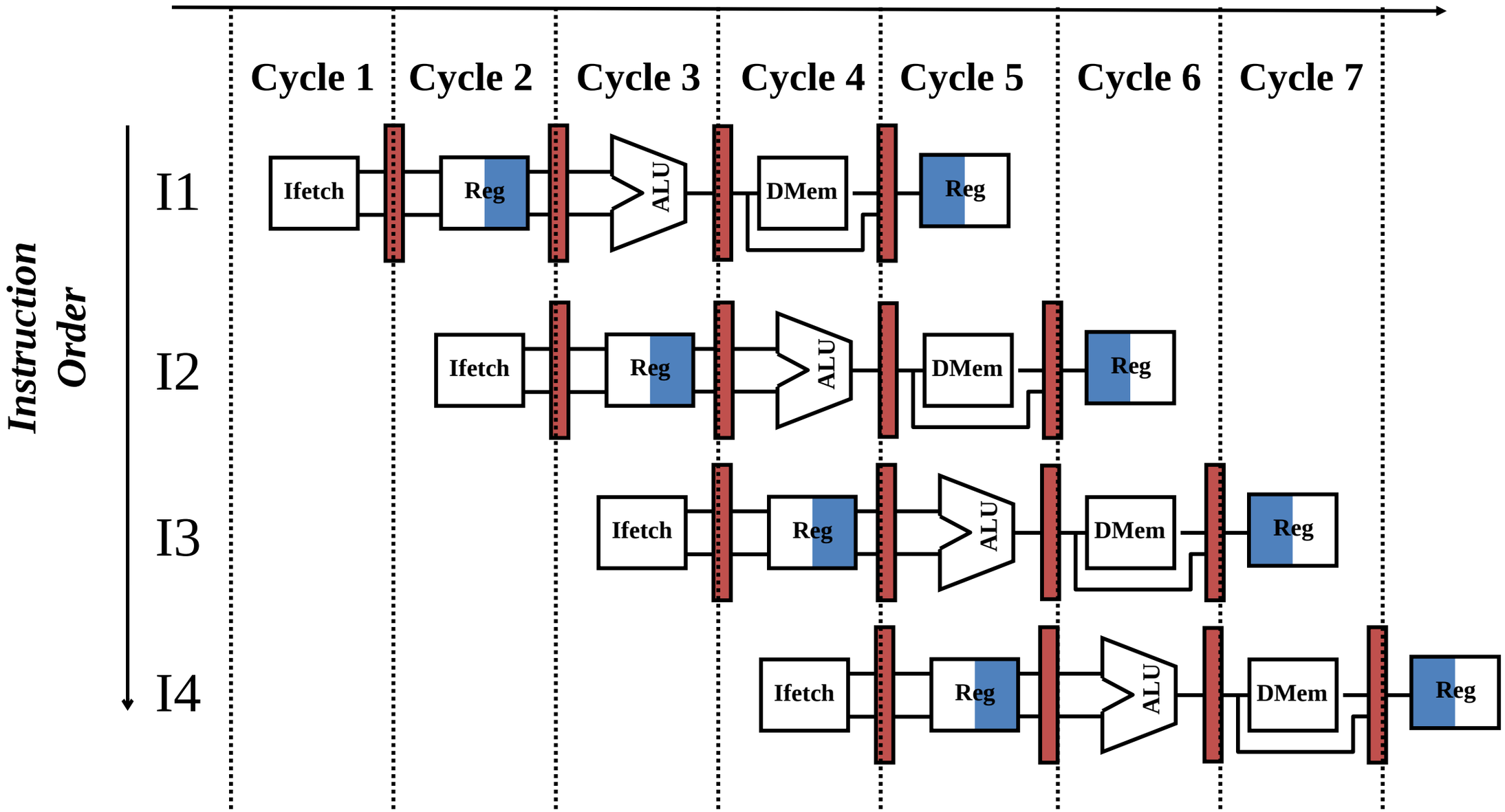
- Flow rate of the pipeline is $1/D$
- Latency of the pipeline is $N*D$

- Ignoring startup and shutdown, the maximum speedup of a pipeline is N

Source: J.N. Amaral

Instruction Execution Pipeline Hardware

- Consider the hardware for instruction execution
  - Resources required for operation



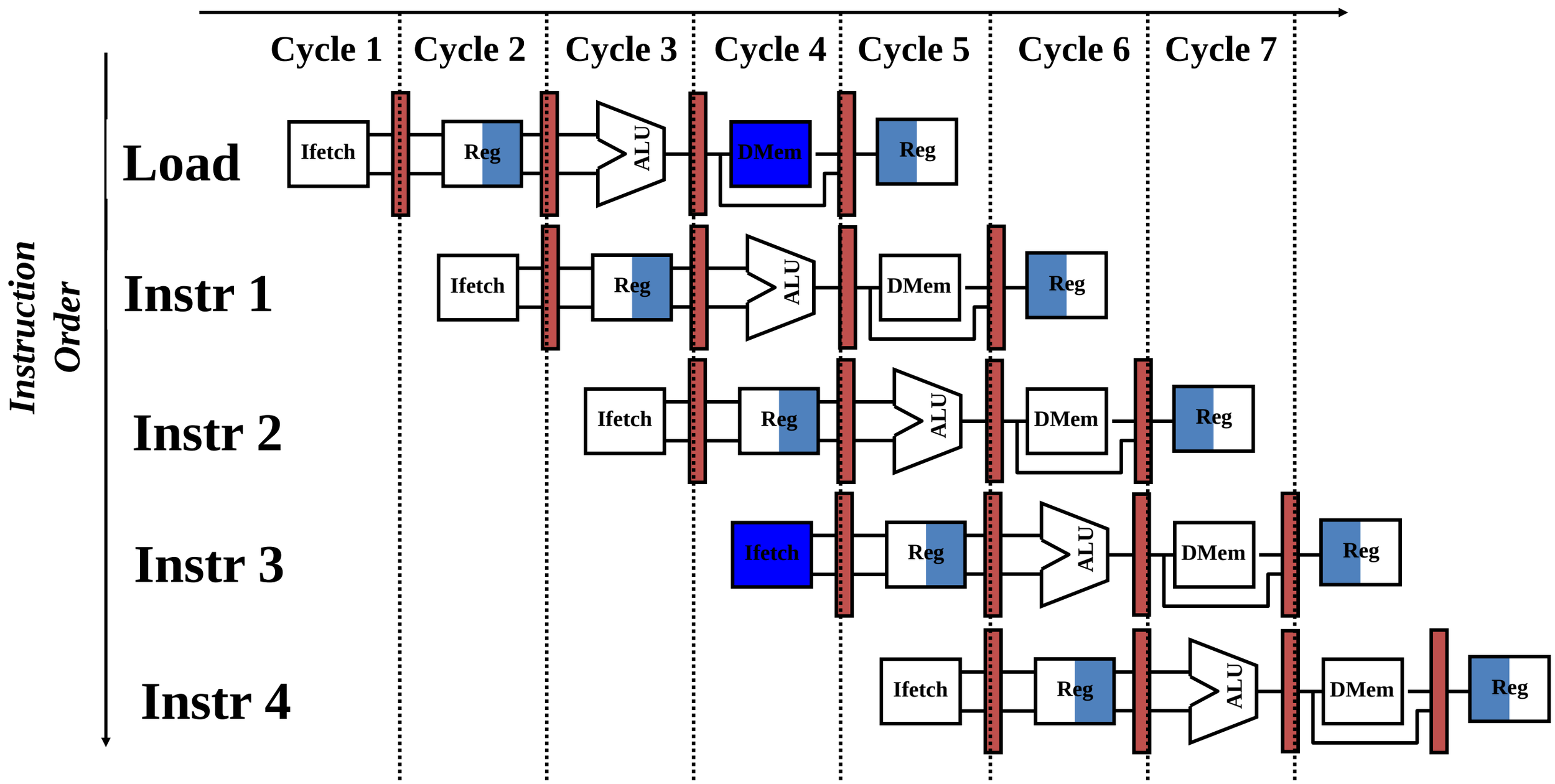
Pipeline Limitations

- Hardware that implements the instruction pipeline must be used for multiple instructions
- Limited hardware resources will place constraints on instruction execution
- Certain situations (*hazards*) can prevent the next statement from executing during its clock cycle
  - *Structural hazards*: hardware can not support a combination of instructions (same hardware resource is accessed)
  - *Data hazards*: execution depends on the result of a previous statement that is still in the pipeline
  - *Control hazard*: loading of instructions is delayed by a decision of change of control flow

Source: J.N. Amaral

Structural Hazards

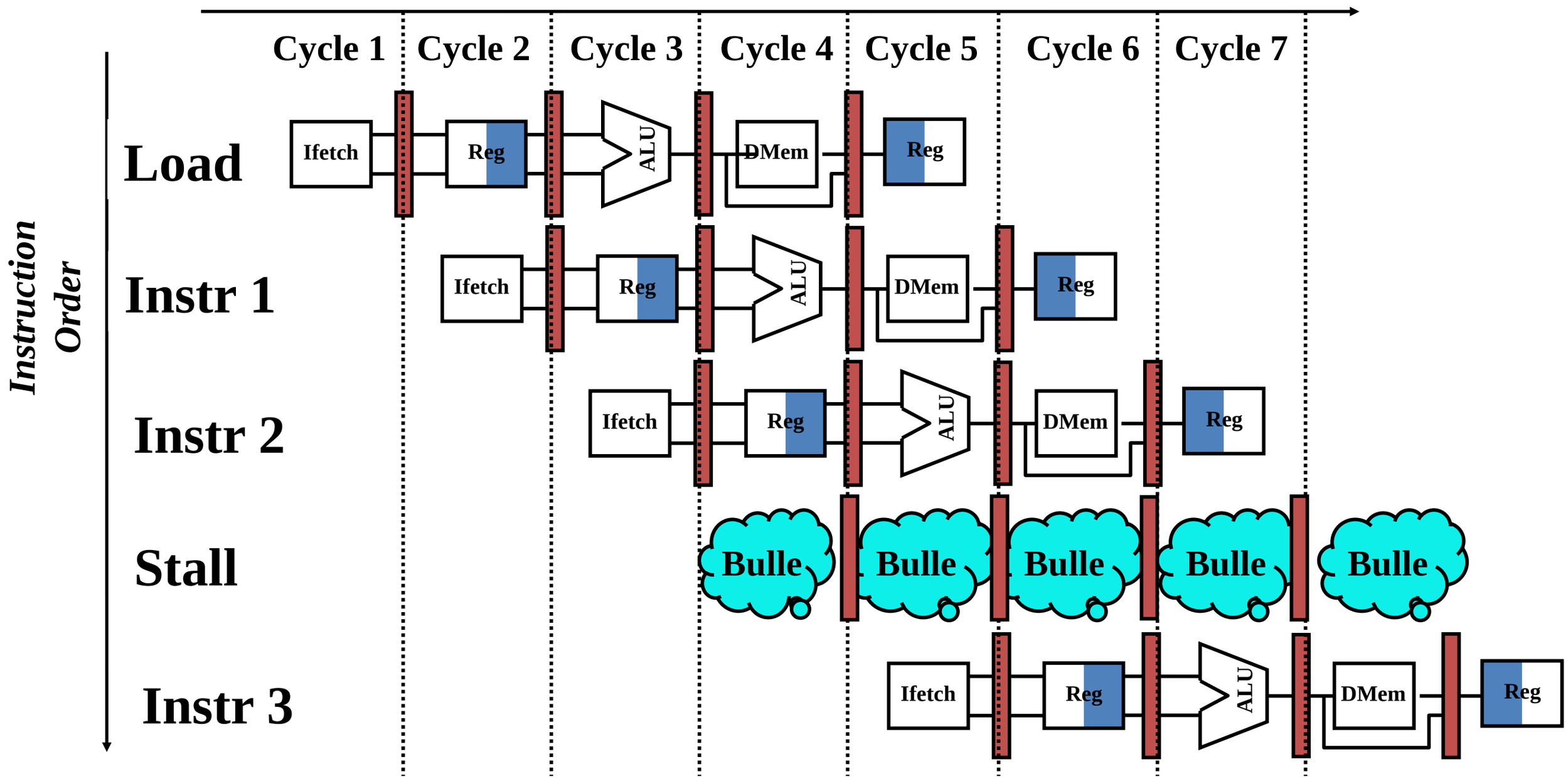
- Instruction fetch and data RW/WR can conflict



Source: J.N. Amaral

Structural Hazards (2)

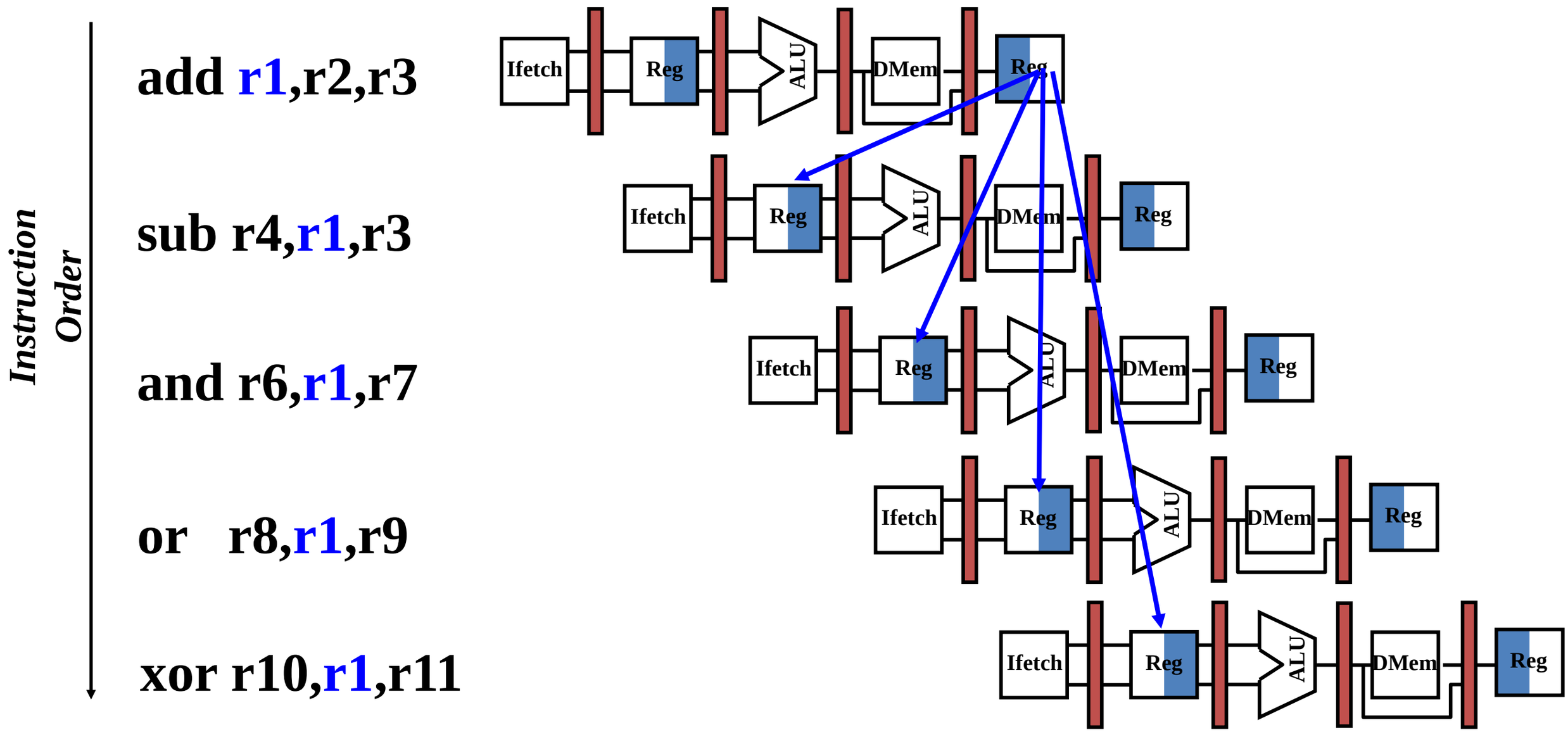
- Hazard causes a stall
  - A “bubble” is inserted and flows through the pipeline



Source: J.N. Amaral

Data Hazards

- CPU registers are used for fast data access
  - # registers is finite and there can be conflicts



Data Dependencies and Pipeline Control

- *Read After Write (RAW)* : flow dependency

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

- *Write After Read (WAR)* : anti-dependency

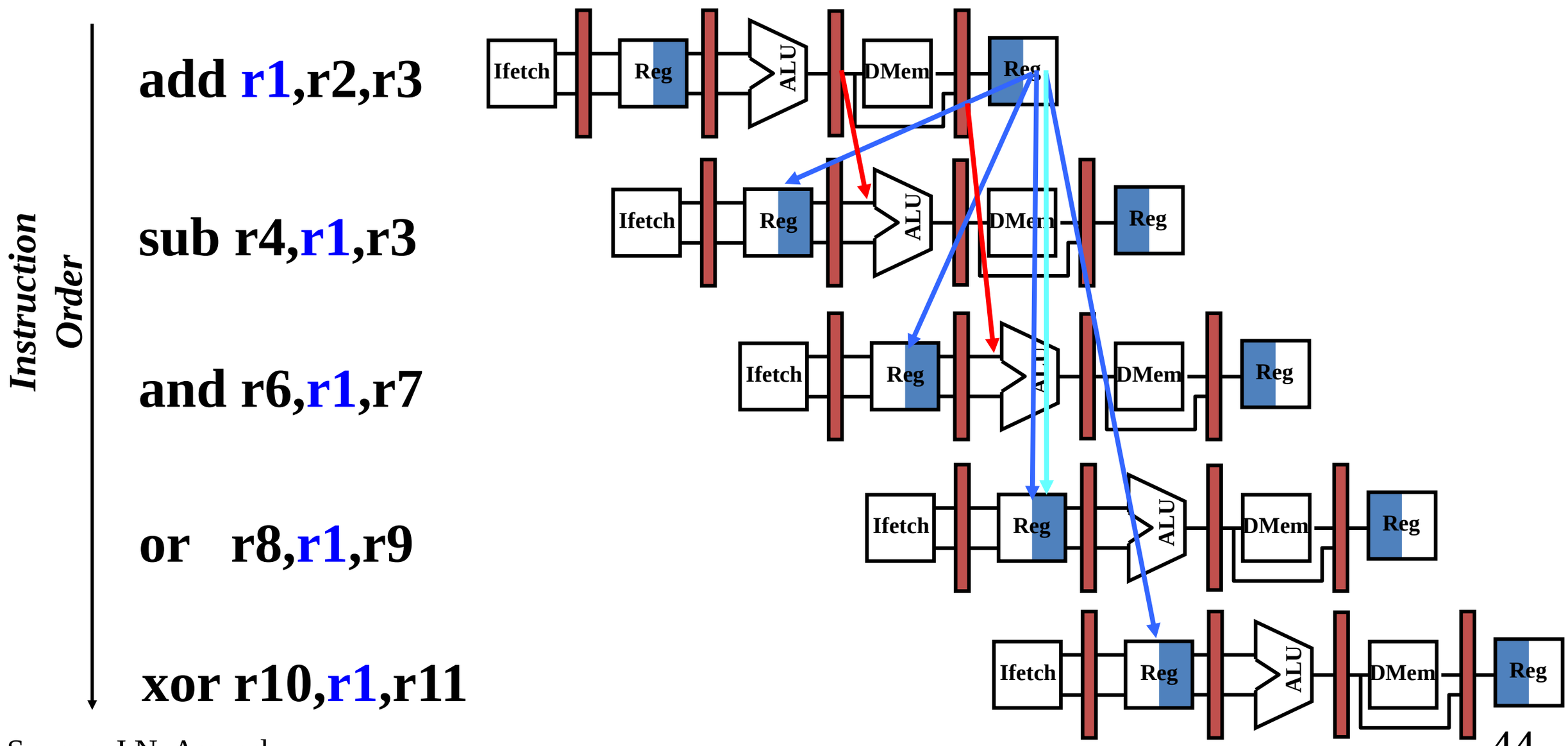
```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- *Write After Write (WAW)* : output dependency

```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

Avoiding Data Hazards w/ Pipeline Control

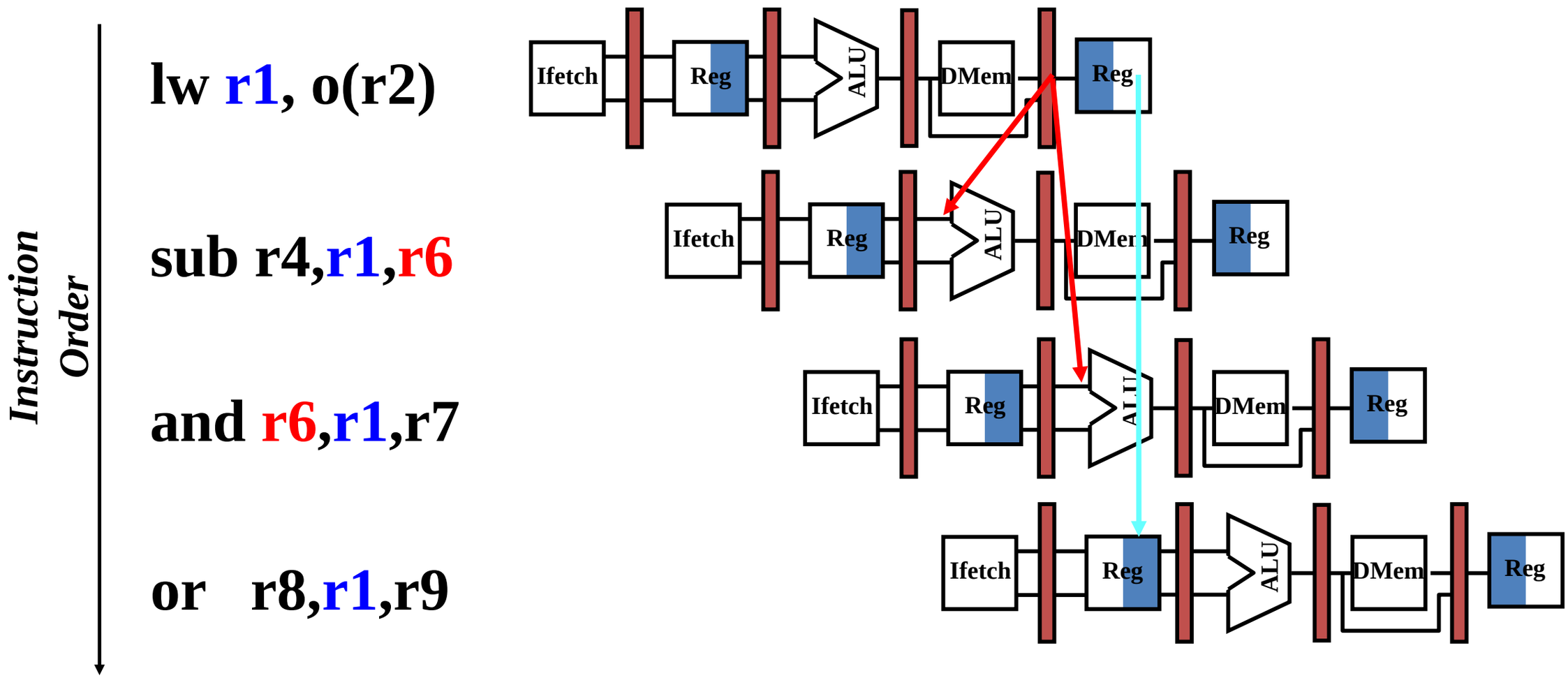
- Short circuit (link) pipeline stages with:
  - *Bypassing* – send value to directly to ALU input
  - *Forwarding* – use 2nd half of register write for read



Source: J.N. Amaral

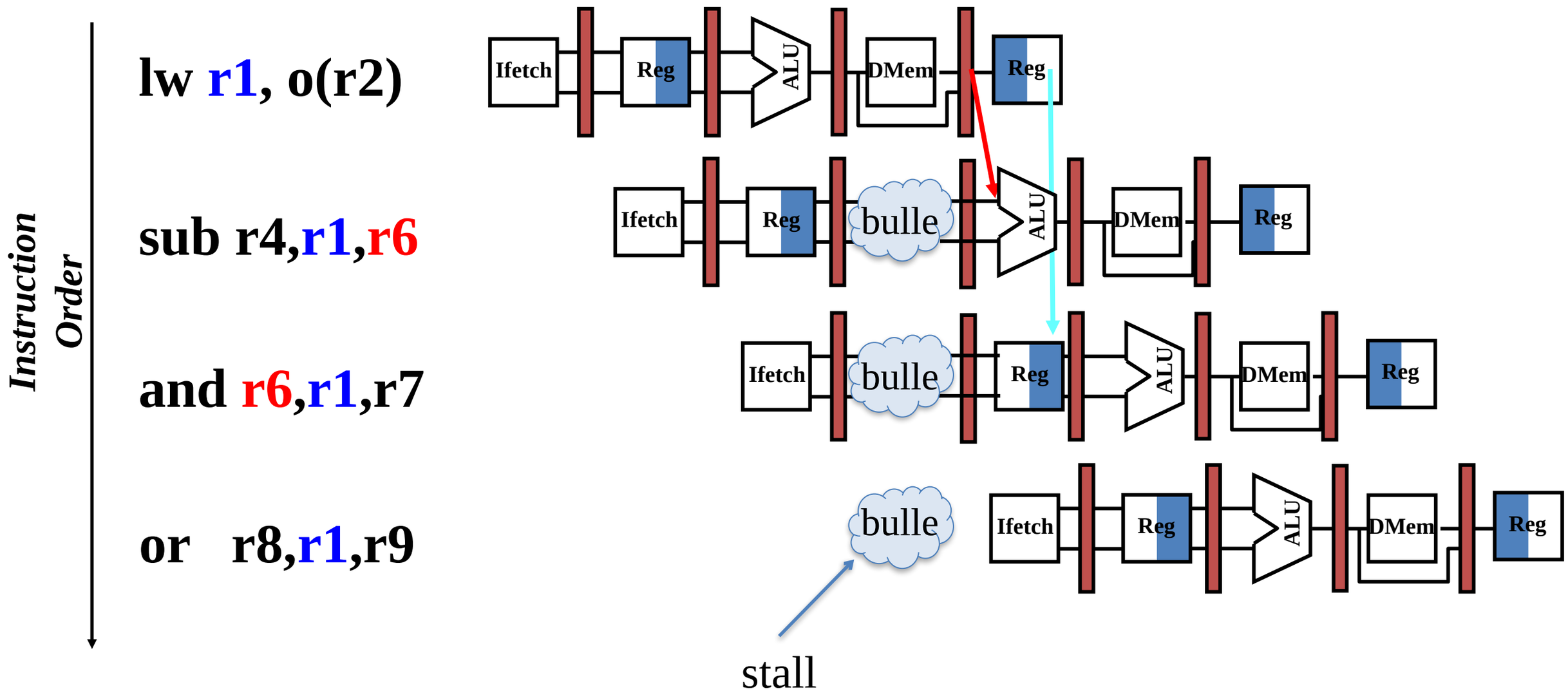
Avoiding Data Hazards (2)

- It might not be possible to bypass or forward without introducing a stall



Avoiding Data Hazards (3)

- Stall shifts the pipeline stages to make bypassing and forwarding possible



Avoid Data Hazards in Code Generation

- Suppose there is the following code:

  a = b + c;

  d = e - f;

  - Assume a, b, c, d, e, f are stored in memory
- Try to generate code

Slow:

```
LW Rb, b
LW Rc, c
ADD Ra, Rb, Rc
SW a, Ra
LW Re, e
LW Rf, f
SUB Rd, Re, Rf
SW d, Rd
```

Faster:

```
LW Rb, b
LW Rc, c
LW Re, e
ADD Ra, Rb, Rc
LW Rf, f
SW a, Ra
SUB Rd, Re, Rf
SW d, Rd
```

Why is this faster?

Try to Avoid Hazards in Code Generation

- Look at the different assembly code and identify the hazards in each and compare

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
LW Rb, b	IF	ID	EX	MEM	WB										
LW Rc, c		IF	ID	EX	MEM	WB									
ADD Ra, Rb, Rc			IF	ID	stall	EX	MEM	WB							
SW a, Ra				IF	stall	ID	EX	MEM	WB						
LW Re, e					stall	IF	ID	EX	MEM	WB					
LW Rf, f							IF	ID	EX	MEM	WB				
SUB Rd, Re, Rf								IF	ID	stall	EX	MEM	WB		
SW d, Rd									IF	stall	ID	EX	MEM	WB	

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
LW Rb, b	IF	ID	EX	MEM	WB										
LW Rc, c		IF	ID	EX	MEM	WB									
LW Re, e			IF	ID	EX	MEM	WB								
ADD Ra, Rb, Rc				IF	ID	EX	MEM	WB							
LW Rf, f					IF	ID	EX	MEM	WB						
SW a, Ra						IF	ID	EX	MEM	WB					
SUB Rd, Re, Rf							IF	ID	EX	MEM	WB				
SW d, Rd								IF	ID	EX	MEM	WB			

Source: J.N. Amaral

Code Scheduling

- The compiler can try to *schedule* the pipeline to avoid stalls due to data hazards
  - Rearranging the code sequence to eliminate the hazard
  - Referred to as *pipeline* or *instruction* scheduling
- Takes into account what is known about how the CPU pipeline hardware operates
- In addition to being useful for data hazards, it is important for control hazards
- This is known as *static scheduling* because it is done before program execution

Pipelines with Multicycle Operations

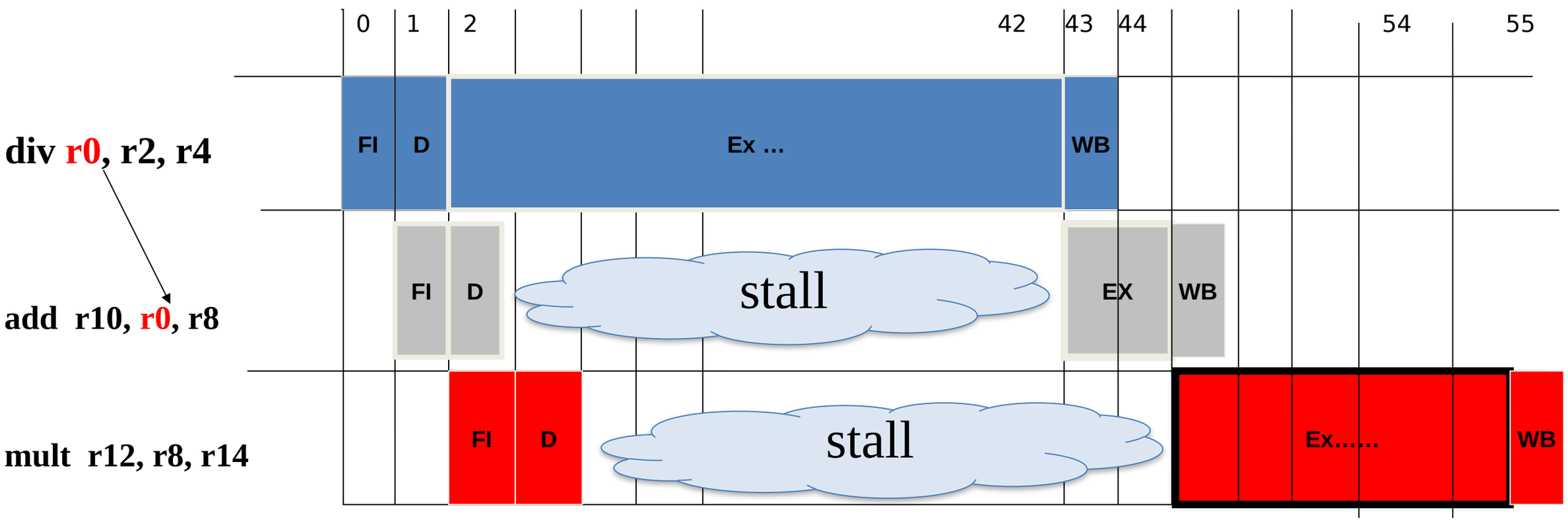
- Some operations (like floating point) are difficult to do in a single clock cycle
  - Often require multiple clock cycles
  - Hardware complexity is more severe
- Multiple functional units can be used
- It is also possible to pipeline a functional unit
  - It is implemented in stages
  - Allows new operations to enter the functional unit as soon as the initial stage is available

Definitions

- *In-order issue* (fetch)
  - All instructions MUST be issued (fetched) in the same order as they appear in the program
- *In-order execution*
  - All instructions MUST enter their execution cycle in the same order as they appear in the program
- *In-order completion*
  - All instructions MUST exit (complete) the pipeline in the same order as they appear in the program
- *Out-of-order execution*
  - Instructions are not required to enter their execution cycle in the same order as they appear in the program
- *Out-of-order completion*
  - Instructions are not required to exit (complete) the pipeline in the same order as they appear in the program

In-Order Instruction Execution

- All instructions MUST execute their pipeline stages in the same order as they appear in the program
- Consider a 4-stage pipeline: IF, D, EX, WB



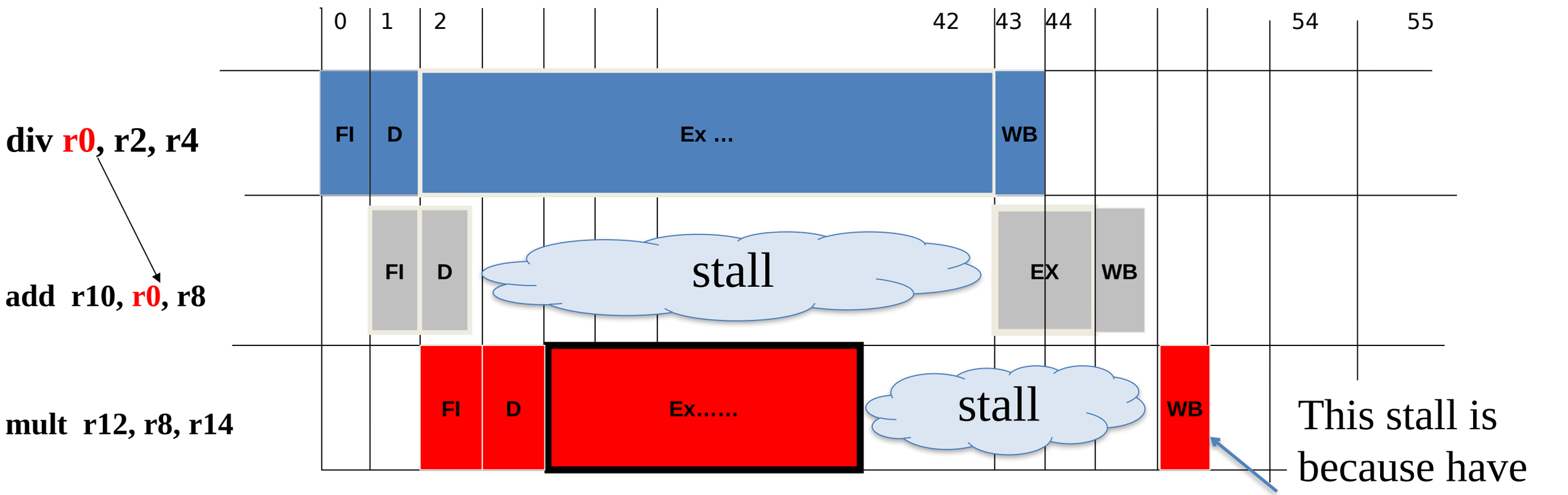
40 cycles div, 2 cycles add , 10 cycles mult

Total execution: 55 cycles

Source: Smail Niar

Out-of-Order Instruction Execution

- Given separate divide and multiply functional units
- It is possible to use both units at the same time
- It must be possible for instructions to execute pipeline stages out-of-order relative to each other



Total execution: 46 cycles (versus 55)

Source: Smail Niar

Data Hazards and Dynamic Scheduling

- If there is a data dependence that cannot be hidden by forwarding, the hazard detection hardware will stall the pipeline
- Static scheduling can help to minimize the actual hazards and resultant stalls
- An early improvement in processor architecture used *dynamic scheduling* to detect and control dependencies during pipeline execution
  - Works with multiple functional resources
  - Manages pipelined, out-of-order instruction execution
  - Significant increase in hardware complexity

Dynamic Scheduling Concept (1)

- If instruction *j* depends on a long-running instruction *i* currently executing in the pipeline, then all instructions after *j* must be stalled until *i* finishes and j can execute.
- Instead, we would like an instruction to begin execution as soon as its data operands are available
- To do so, the pipeline must be able to do out-of-order execution
- Out-of-order execution => out-of-order completion

Dynamic Scheduling Concept (2)

- Split ID into 2 stages:
  - *Issue*: decode instruction, check for structural hazards
  - *Read operands*: wait until no data hazards, then read operands
- Distinguish between when instruction *begins execution* and when it *completes execution*
  - Inbetween these two the instruction is *in execution*
- Given hardware support, multiple instructions can be in execution at the same time

Dynamic Scheduling with a Scoreboard

- In a dynamically scheduled pipeline, all instructions pass through the *issue* stage in-order
- However, they can be stalled or bypass each other in the *read operands* stage
- A technique called *scoreboarding* allows instructions to execute out-of-order when there are:
  - Sufficient resources (there can be multiple resources)
  - No data dependences (all operands are available)
- The goal of the scoreboard is to achieve a CPI=1 when there are no structural hazards

Scoreboarding Operation

- Scoreboarding takes full responsibility for instruction issue and execution, including all hazard detection
- Multiple functional units are needed since multiple instructions will be in their execution stage at the same time
- Scoreboard keeps a dynamic record of data dependencies
- Scoreboard determines when the instruction can read its operands and begin execution
- If an instruction cannot execute immediately, the scoreboard monitors every change in the HW and decides when the instruction can execute
- Scoreboard controls when an instruction can write its result to a destination register

Three Parts of a Scoreboard

- Instruction status
  - Which of the 4 steps the instruction is in
- Functional unit status
  - Indicates the state of the functional units
  - *Busy*: indicates whether the unit is busy or not
  - *Op*: operation to perform on the unit
  - *Fi*: destination register
  - *Fj*, *Fk*: source registers
  - *Qj*, *Qk*: functional units producing *Fj*, *Fk*
  - *Rj*, *Rk*: flags indicating when *Fj*, *Fk* are ready
- Register result status
  - Indicates which functional unit will write each register

Scoreboard Factors

- A scoreboard is limited by:
  - Amount of instruction independence
    - *true* data dependencies can not be avoided
  - Number of scoreboard entries
    - this determines how far ahead the pipeline can look for independent instructions
    - this set of instructions is call the *window*
  - Number and types of functional units
  - Presence of anti- (WAR) and output (WAW) dependencies

CDC 6600 Scoreboard

- The CDC 6600 was the first machine to conceive of using a scoreboard
- Organization
  - 16 functional units
    - 4 FP, 5 Mem, 7 Int
  - Load-store memory
    - not registers!
  - 4 groups of functional units each with a data trunk
  - Only 1 unit in a group could use the data trunk at a time
- CDC 6600 was a very early machine

Tomasulo Algorithm

- CPU architecture for out-of-order execution
- Hardware technique developed by Robert Tomasulo in the 1960's at IBM for the 360/91
  - Improved on the notion of *scoreboarding* as a technique to keep track of dependencies and functional unit use

- Key ideas
  - Treat memory operations as functional units
  - Abstract functional units to *reservation stations*
  - Renaming to eliminate false dependencies
- Limited hardware (no caches, small # FP registers)

Tomasulo Organization

- Each instruction is divided into 3 steps
  - *Issue*: taking instruction into account and waiting for input operands
  - *Execution*: Effective execution of the operation on a functional unit
  - *Write*: writing the result and passing information to the following instructions
- Technology based on a *Common Data Bus* (CDB) for the transfer of information
- Use of reservation stations

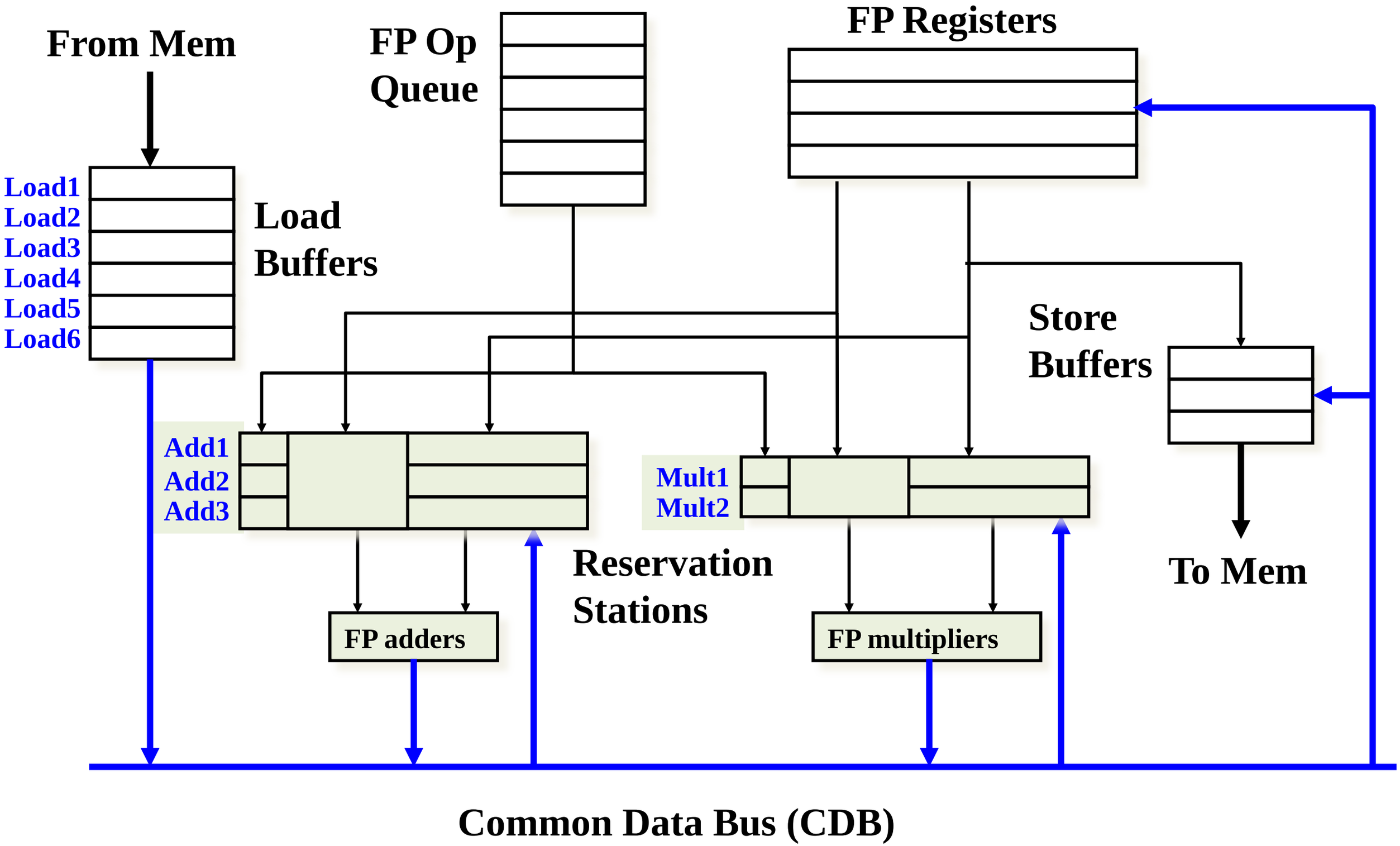
Reservation Stations

- Abstract functional units by adding input buffers
  - Can store more instructions waiting for an operand
- Components of a station = an instruction
  - *Busy*: Is this station busy?
  - *Op*: Assembly operation using this station
- Management of the operands for instruction
  - *Vj*, *Vk*: value of the operands already present
  - *Qj*, *Qk*: name of the reservation station that will produce the operand

Register Bank Extension

- Register bank contains all the processor registers
- Addition of a flag to know which reservation station will produce a value *(Qi)*
- Result
  - Dynamic renaming of registers
  - Suppression of false dependencies (WAR and WAW)

Tomasulo Organization



Tomasulo Example

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2			
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time Name	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
Add1	No					
Add2	No					
Add3	No					
Mult1	No					
Mult2	No					

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
0	*FU*									

Source: D. Culler

Tomasulo Example Cycle 1

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1		
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	No	
Load3	No	

Reservation Stations:

Time Name	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
Add1	No					
Add2	No					
Add3	No					
Mult1	No					
Mult2	No					

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
1	*FU*				Load1					

Source: D. Culler

Tomasulo Example Cycle 2

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1		
LD	F2	45+	R3	2		
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
2	*FU*		Load2		Load1					

Note: Can have multiple loads outstanding

Source: D. Culler

Tomasulo Example Cycle 3

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	
LD	F2	45+	R3	2		
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
3	*FU*	Mult1	Load2		Load1					

Note: registers names are removed ("renamed") in Reservation Stations; MULT issued

Load1 completing; what is waiting for Load1?

Source: D. Culler

Tomasulo Example Cycle 4

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	Yes	SUBD	M(A1)			Load2
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
4	*FU*	Mult1	Load2		M(A1)	Add1				

Load2 completing; what is waiting for Load2?

Source: D. Culler

Tomasulo Example Cycle 5

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
2	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	No					
	Add3	No					
10	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
5	*FU*	Mult1	M(A2)		M(A1)	Add1	Mult2			

Timer starts down for Add1, Mult1

Source: D. Culler

Tomasulo Example Cycle 6

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
1	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
9	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
6	*FU*	Mult1	M(A2)		Add2	Add1	Mult2			

Issue ADDD here despite name dependency on F6?

Source: D. Culler

Tomasulo Example Cycle 7

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
0	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
8	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
7	*FU*	Mult1	M(A2)		Add2	Add1	Mult2			

Add1 (SUBD) completing; what is waiting for it?

Source: D. Culler

Tomasulo Example Cycle 8

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
2	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
7	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock **8**

	F0	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
FU	Mult1	M(A2)		Add2	(M-M)	Mult2			

Source: D. Culler

Tomasulo Example Cycle 9

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
1	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
6	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
9	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

Source: D. Culler

Tomasulo Example Cycle 10

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
0	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
5	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
10	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

Add2 (ADDD) completing; what is waiting for it?

Source: D. Culler

Tomasulo Example Cycle 11

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

	Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
		Add1	No					
		Add2	No					
		Add3	No					
	4	Mult1	Yes	MULTD	M(A2)	R(F4)		
		Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
11	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Write result of ADDD here?

All quick instructions complete in this cycle!

Source: D. Culler

Tomasulo Example Cycle 12

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

	Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
		Add1	No					
		Add2	No					
		Add3	No					
	3	Mult1	Yes	MULTD	M(A2)	R(F4)		
		Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
12	FU	Mult1	M(A2)		(M-M+N	(M-M)	Mult2			

Source: D. Culler

Tomasulo Example Cycle 13

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

	Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
		Add1	No					
		Add2	No					
		Add3	No					
	2	Mult1	Yes	MULTD	M(A2)	R(F4)		
		Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
13	FU	Mult1	M(A2)		(M-M+N	(M-M)	Mult2			

Source: D. Culler

Tomasulo Example Cycle 14

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
1	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
14	FU	Mult1	M(A2)		(M-M+N	(M-M)	Mult2			

Source: D. Culler

Tomasulo Example Cycle 15

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
15	*FU*	Mult1	M(A2)		(M-M+N	(M-M)	Mult2			

Mult1 (MULTD) completing; what is waiting for it?

Source: D. Culler

Tomasulo Example Cycle 16

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
40	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
16	*FU*	M*F4	M(A2)		(M-M+N	(M-M)	Mult2			

Just waiting for Mult2 (DIVD) to complete

Source: D. Culler

Tomasulo Example Cycle 55

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
1	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
55	*FU*	M*F4	M(A2)		(M-M+N	(M-M)	Mult2			

Source: D. Culler

Tomasulo Example Cycle 56

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
0	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
56	*FU*	M*F4	M(A2)		(M-M+N	(M-M)	Mult2			

Mult2 (DIVD) is completing; what is waiting for it?

Source: D. Culler

Tomasulo Example Cycle 57

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	57
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
56	*FU*	M*F4	M(A2)		(M-M+N	(M-M)	Result			

Once again: In-order issue, out-of-order execution and out-of-order completion.

Source: D. Culler

Tomasulo Benefits and Drawbacks

- Benefits
  - Register renaming effectively allowed building data flow dependency graph on the fly
  - Elimination of stalls for WAW and WAR hazards
  - Effectively achieves dynamic loop unrolling
  - Reservation stations overcome limitations of pure scoreboarding
- Drawbacks
  - Hardware complexity
  - Many associative stores (CDB) at high speed
  - Performance limited by Common Data Bus
    - each CDB must go to multiple functional units
    - completion of functional units limited to one per cycle
  - Non-precise

Instruction-Level Parallelism (ILP)

- Instruction-level parallelism (ILP) is the overlapped execution of instructions (i.e., execution of instructions in parallel)

$$CPI_{pipe} = CPI_{ideal} + Stalls_{structural} + Stalls_{RAW} + Stalls_{WAR} + Stalls_{WAW} + Stalls_{control}$$

- Exploit independence among instruction sequences
- Amount of parallelism available within a basic block is quite small
- Must exploit ILP across multiple basic blocks
- Control hazards are serious limitations to ILP

Consider a Code with Branches

- If the branch is taken, 3 cycles are lost
- Question is which control path is more likely

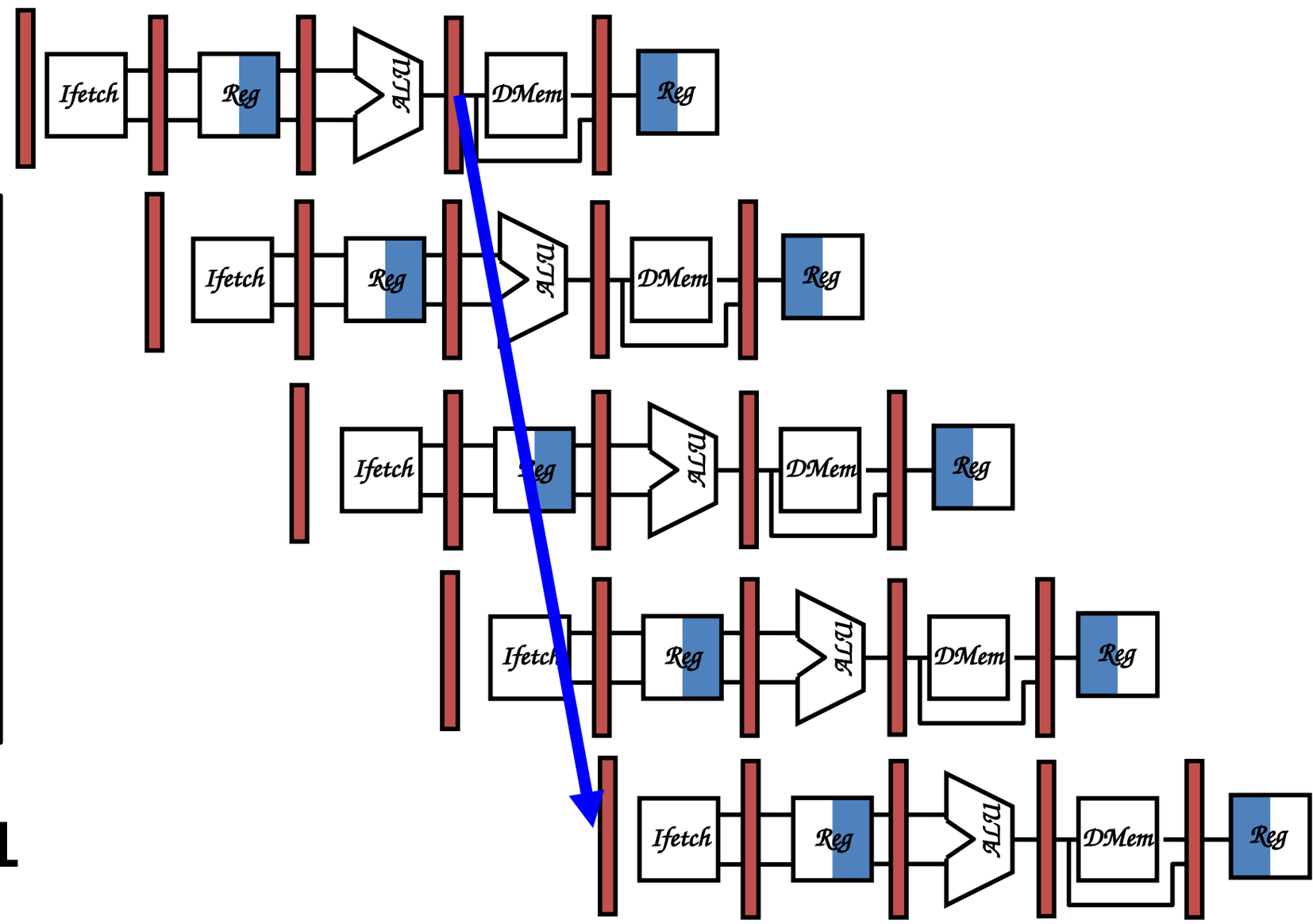
```
10: beq r1,r3,36

14: and r2,r3,r5

18: or  r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11
```



Source: J.N. Amaral

Branch Penalties

- If 30% of control decisions is to branch, a penalty of 3 cycles is not negligible
- Two-part solution:
  - Determine early if the branch it to me made
  - Calculate the branch target address as soon as possible
- Typically, a branch tests a register (= 0 or ≠ 0)
- Possible approach:
  - Test on zero is moved to the ID / RF stage
  - Addition of an adder to calculate the new PC in the ID / RF stage

Four Approaches for Control Hazards (1)

- Approach 1: Wait until the branch direction is identified
- Approach 2: Predict that the branch is not taken
  - Execute successive instructions in sequence
  - Purge instructions in the pipeline if the branch was made
  - Runs instructions in sequence is more efficient
- Approach 3: Predict that the connection would be made
  - Branch would not have yet calculated their target address
  - Results in a 1 cycle penalty

Source: J.N. Amaral

Four Approaches for Control Hazards (2)

- Approach 4: Delay the branch
  - Let *N* instructions execute after the branch instruction
  - Then branch if necessary
  - Violate the sequential semantics

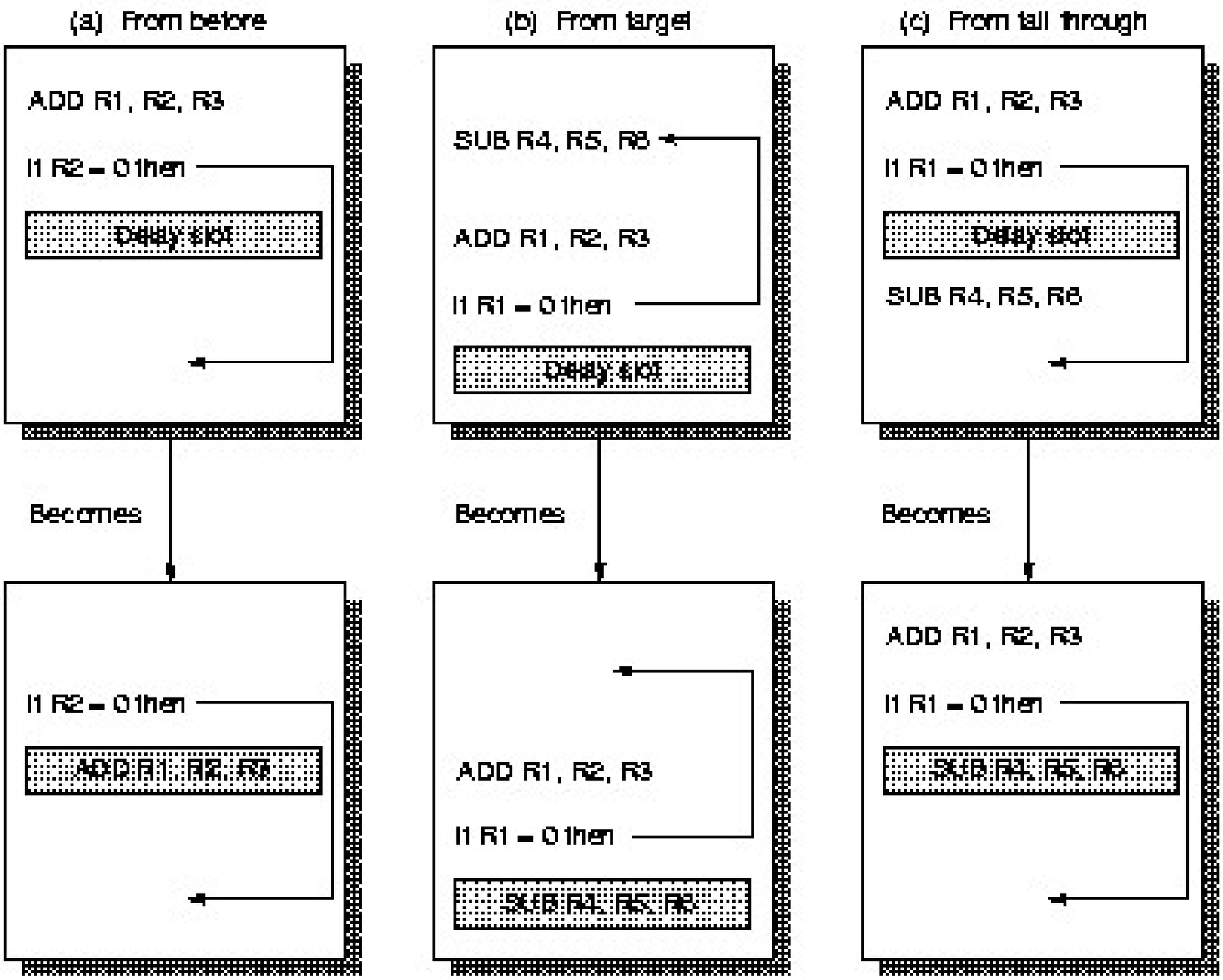
Source: J.N. Amaral

Delayed Branches (1)

- The compiler or programmer must find N instructions to fully exploit the delay
  - Filling the *delay slot*
- Where can these instructions be found?
  - Take the n instructions that precede it sequentially
  - Take N instructions from the target address
    - must be only for unconditional, or speculated as taken
  - Take an instruction that follows it

Source: J.N. Amaral

Delayed Branches (2)



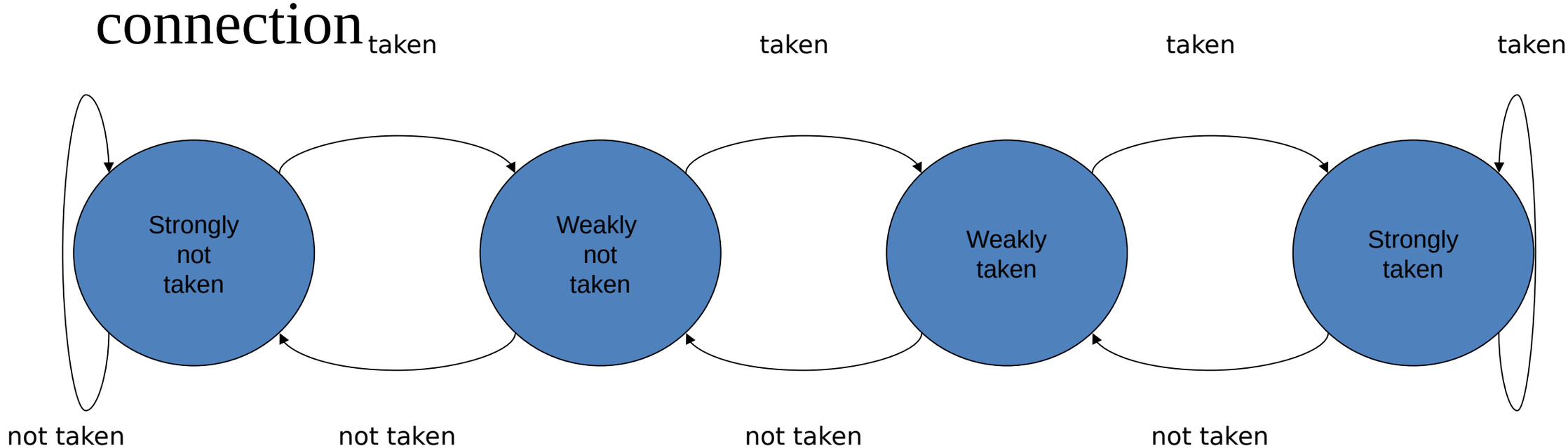
Source: J.N. Amaral

Branch Prediction (1)

- Most used solution on modern processors
- Idea is predict the result of the branch (outcome) taking into account two indications
  - History of the same branch
  - History of the last branch encountered
- Anticipates if a branch is going to be taken or not
- Suppose, for example
  - 98% of connections correctly predicted
  - 2% will be expensive in terms of cycles lost, but they are infrequent

Branch Prediction (2)

- Finite state automaton for predicting a branch
- Consider a 2-bit counter
  - 2-bit saturating up-down counter
  - Change prediction with misprediction in *Weakly* state
  - Predict a branch not taken in *not taken* states
  - Predict a branch taken in *taken* states
  - Update of the current state according to the actual result of the connection



Branch Prediction (3)

- Adding a branch history level
- Storage in *Branch History Register* (BHR)
- Whenever a branch is encountered and decided
  - Shift the BHR to the left one bit
  - Addition of a low 0 if this branch has not been taken
  - Added a 1 otherwise

Two-level Branch Prediction

- Use BHR as an index in a *prediction history table (PHT)*
  - Each entry in the PHT contains a 2-bit counter
  - Basic principle used in modern HPC processors
- Two-level prediction scheme
  - Read the contents of the BHR
  - Access to PHT based on the BHR index
  - Predict the branch according to the associated counter
  - Once the branch result is known, update the BHR and the associated counter
- For more details :
  - Yeh and Patt, *Two-Level Adaptive Training Branch Prediction*, MICRO 1991
  - Yeh and Patt, *Alternative Implementation of Two-Level Adaptive Branch Prediction*, ISCA 1992

Speculative Execution

- Allows the processor to execute instructions beyond the branches speculatively
- Then, if the branch was not correctly predicted, the execution of the speculated operations is canceled (*pipeline dump*)
- If the branch was correctly predicted, the results of the speculated operations are validated in the architectural registers (last step of the pipeline)

Predicates

- Branch predictors and speculative execution are micro-architectural techniques
- Using predicates is an architectural technique
- Idea is to associate boolean registers (predicates) with assembler instructions
- The processor pipeline executes predicated statements regardless of the value of the predicates
- However, the pipeline validates or not the results of the execution according to the value of the predicate
- Predicates are set as a result of conditional statements

Limitations

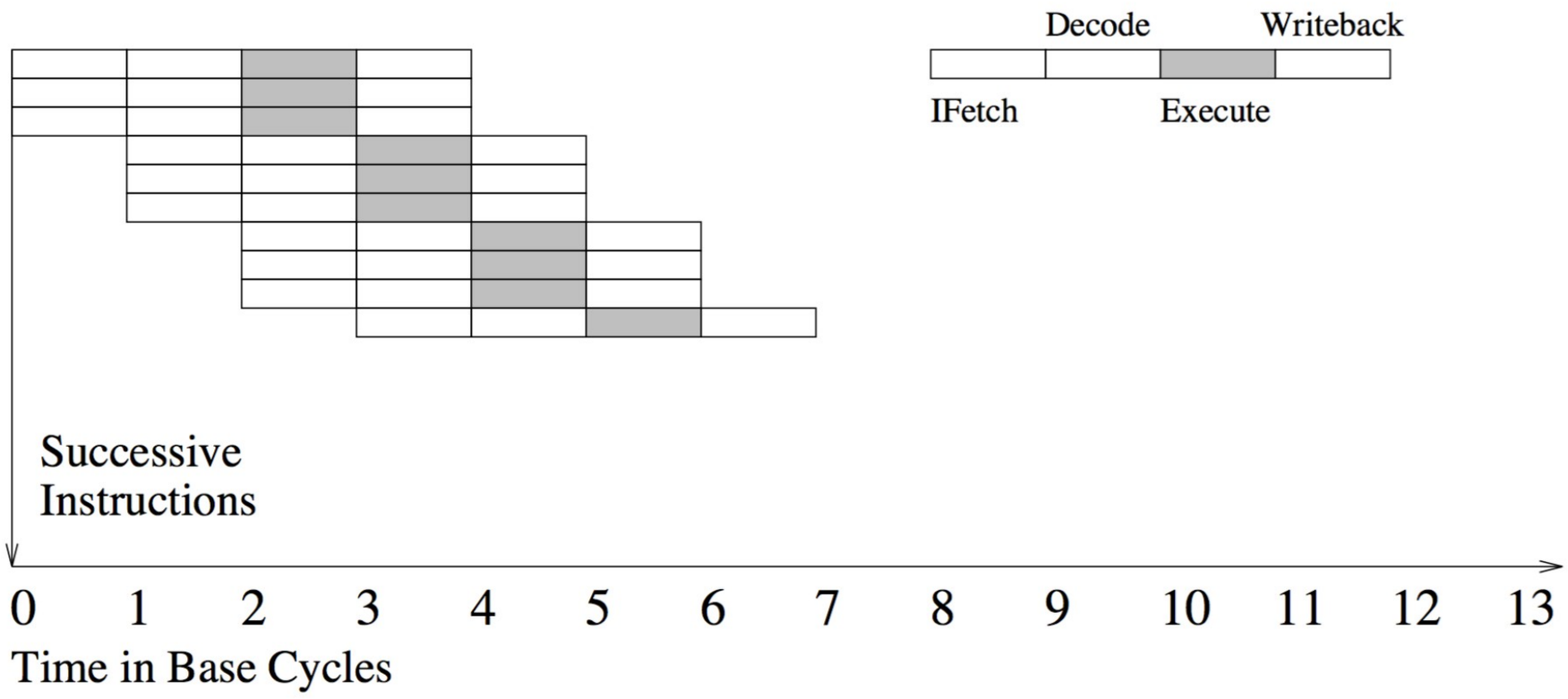
- Maximum rate of one instruction per cycle (IPC = 1)
  - Despite micro-architectural optimizations described above
- In practice, several concerns
  - Conflicts over functional units,
  - Memory access latency
  - Data Dependencies
  - Branches
- Solutions:
  - Temporary storage of the last data accessed in a fast memory (cache and memory hierarchy)
  - Execute multiple instructions at a time

Instruction-Level Parallelism

- Where can we find independent instructions (or independent portions of) to execute?
  - Opportunities for splitting up instruction processing and executing multiple parts of instructions simultaneously
- Ideas so far:
  - Pipelining within instruction and between instructions
  - Overlapped execution and multiple functional units
  - Out of order execution
- Other ideas to consider:
  - Multi-issue execution
  - Superscalar processing
  - Superpipelining
  - Superscalar superpipelining
  - Very Long Instruction Word (VLIW)
  - Hardware multithreading (hyperthreading)

Superscalar Processor Architecture

- A superscalar CPU of degree *N* can issue *N* instructions per cycle
  - Called *multi-issue* execution
- In order to fully utilize a superscalar machine of degree *N* there must be *N* instructions that can execute at all times
- More complex hardware is clearly required to support all the instructions executing
- Superscalar machine were originally conceived of as alternatives to vector machine



Optimizations for Superscalar Processors

- Superscalar processors have a limited window of dynamic optimization
- Limitations of ILP
- Must rely more on the compiler for better IPC
  - Compiler and / or programmer has a more global view
- Superscalar can benefit from dynamic scheduling
- Combination of compiler and hardware can achieve good IPC performance

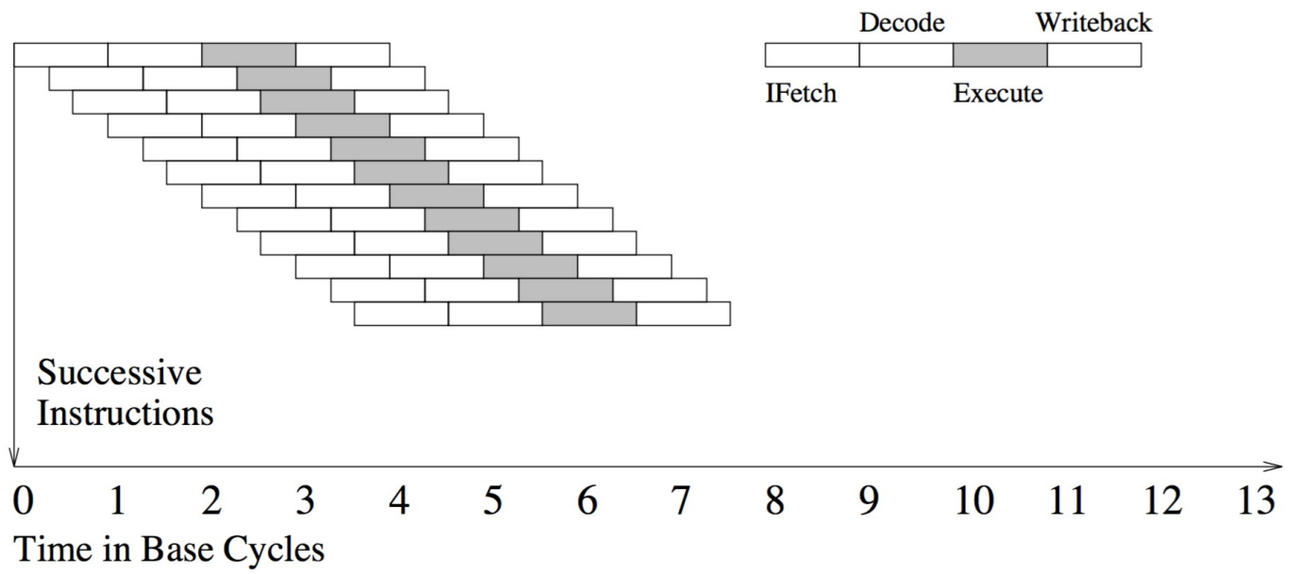
Limitations of Superscalar Processors

- Scalable but limited architectures
  - Ports on registers
  - Pipeline Depth
  - Degrees of parallelism (max current issue = 6)
  - Limits of parallelism in a bounded window
  - High cost
  - Energetic efficiency
- There was a migration from superscalar to multiple cores, hardware multithreading, and so on

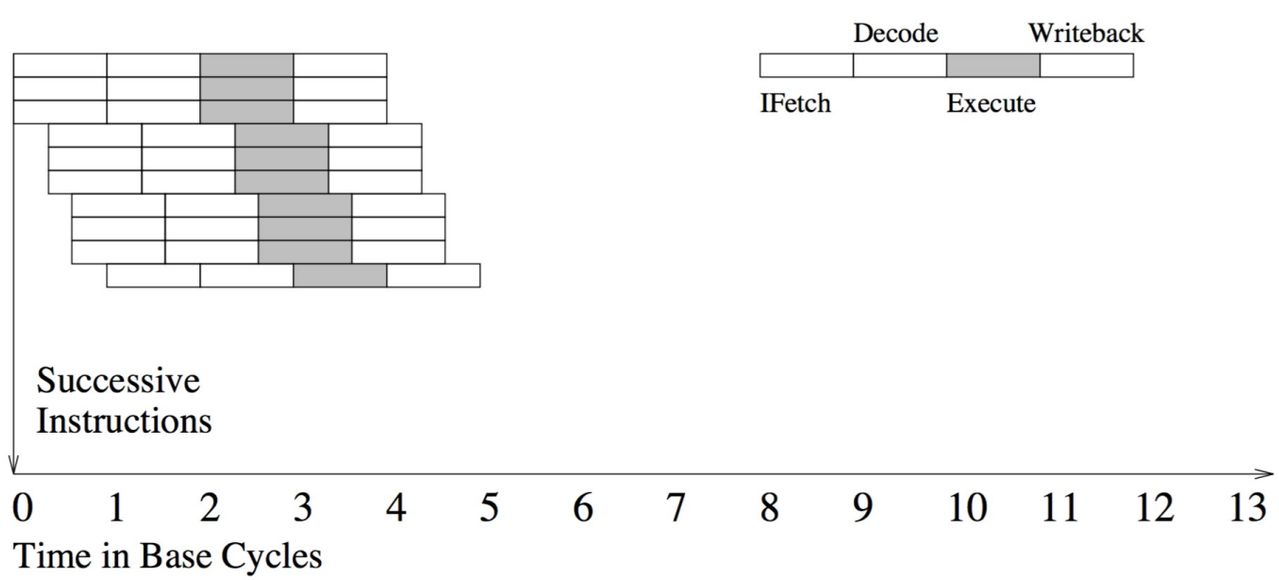
Superpipeline and Superscalar Superpipeline

- Try to reduce the cycle time by dividing each pipeline stage into multiple sub-stages
- Can combine with superscalar

superpipeline



superpipeline
superscalar



Compilers Can Help

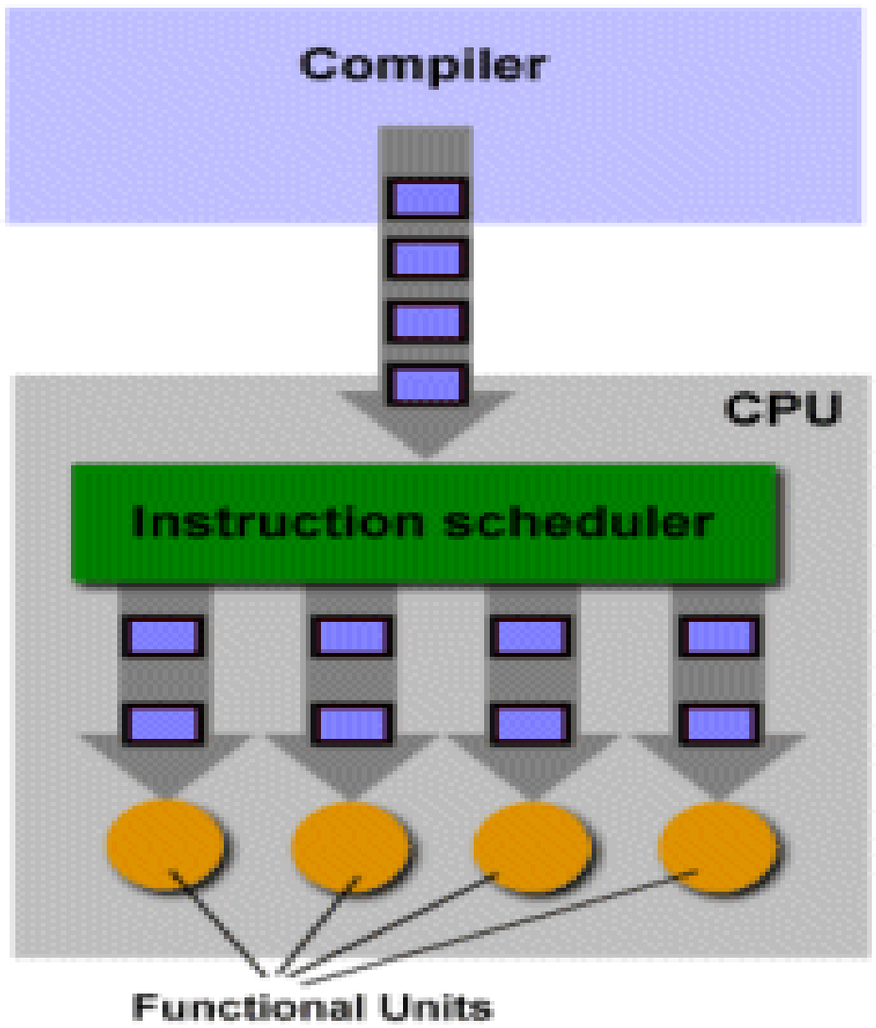
- It can extract parallelism for "well written" codes
- Can schedule instructions in a global window
- Can make a register allocation more intelligent, less aggressive, favoring the extraction of parallelism
- Can generate code with speculative execution
- Can place data, restructure loops to improve spatial and temporal locations of programs (improve caches usage)
- Programmers needs to know the limitations and strengths of optimizing compilers to be able to write effective programs
- Compilers + architecture innovations

VLIW Philosophy

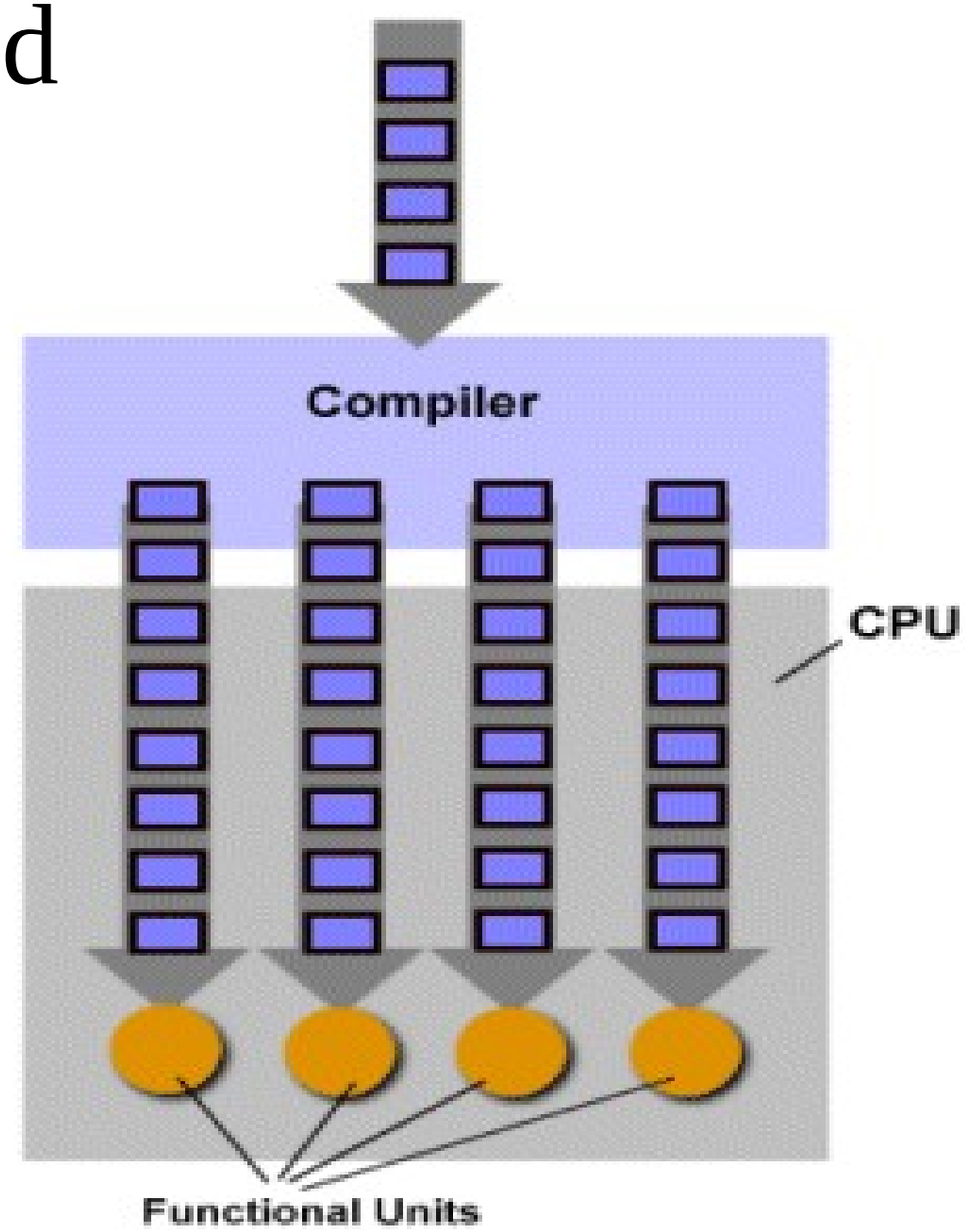
- Let the compiler do its best
  - Optimize as much as it can
- Design a processor that exposes the parallelism at the architectural level
  - Compiled assembler code encodes the parallel packets
- Packets get put in a *very long instruction word* (VLIW)
  - Instruction becomes a concatenation of *n* parallel operations
  - Fixed size
  - Fixed latencies of explicit and non-unitary operations
  - Placement of operations is restricted by the functional units
- Intel Itanium processor family based on VLIW concepts

Superscalar versus VLIW

- VLIW hardware has more flexibility in how instructions can be arranged



Dynamic Superscalar Instruction Scheduling



VLIW Instruction Scheduling

Sequential Code versus VLIW

```
ldw  r1, 0(r2)
sub  r4,r1,r6
and  r6,r1,r7
or   r8,r1,r9
```

```
Cycle 0: ldw    r1, 0(r2)
Cycle 1: …
Cycle 2: …
Cycle 3: sub r4,r1,r6 || and r6,r1,r7
Cycle 4: or r8,r1,r9
```

Sequential code

Superscalar execution

Inst 0:	**ldw r1, 0(r2)**	**; nop**	**; nop**	**; nop;;**
Inst 1:	**nop**	**; nop**	**; nop**	**; nop;;**
Inst 2:	**nop**	**; nop**	**; nop**	**; nop;;**
Inst 3:	**nop**	**; sub r4,r1,r6**	**; and r6,r1,r7**	**; nop;;**
Inst 4:	**nop**	**; or r8,r1,r9**	**; nop**	**; nop;;**

Code VLIW

VLIW format

Scheduling Models

- "Equals" Model (EQ)
  - Each operation lasts exactly its latency
  - For example, the destination register will only be updated when the latency of the operation expires
  - Decreases pressure on registers (better reuse of architectural registers)
  - The exception model becomes complex

- "Less-Than-or-Equals" Model (LEQ)
  - An operation may last less than its latency
  - For example, the destination register can be updated at any time between the launch date and the latency of the operation
  - Simplifies precise management of exceptions
  -

Source : Josh Fisher

Semantics of EQ and LEQ

R1 is considered free here

```
Inst 0:  lw r1, 0(r2); nop              ; nop;              nop;
Inst 1:  nop           ; nop              ; nop;              nop;
Inst 2:  nop           ; nop              ; nop;              nop;
Inst 3:  nop           ; sub r4,r1,r6     ; and r6,r1,r7;     nop;
Inst 4:  nop           ; or r8,r1,r9      ; nop; nop;
```

VLIW code with EQ semantics

```
Inst 0:  lw r1, 0(r2); nop              ; nop;              nop;
Inst 1:  nop           ; sub r4,r1,r6     ; and r6,r1,r7;     nop;
Inst 2:  nop           ; or r8,r1,r9      ; nop;              nop;
```

VLIW code with LEQ semantics

Advantages of VLIW

- Clean, simple and more scalable architecture
- High degree of parallelism of instructions
- Low cost for micro-architecture
- Dynamic executions are visible architecturally
  - More predictable codes

Disadvantages of VLIW

- Works well only for programs with statically extractable parallelism
  - Requires a very powerful optimizing compiler
- Very sensitive to cache layout
- Code size generated can have a significant number of NOPs
- No binary compatibility between generations of the same family

Compacted VLIW Instructions

Opc Dst Src1 Src2	Opc Dst Src1 Src2	Opc Dst Src1 Src2	Opc Dst Src1 Src2
Operation 1	**Operation 2**	**Operation 3**	**Operation 4**

- If not compacted, the VLIW instructions are wide ...
- Compilers can not always fill all holes
  - Functional units and latencies
- Code space is not free
  - Embedded, it is even very expensive
- How to reduce code size?
  - Remove NOPs
  - Use a compact format
- With variable VLIW instruction encoding (stop-bit)
- Compact code

Source : Josh Fisher

Differences Between Superscalar and VLIW

	Superscalar	VLIW
Instruction flows	The instructions are loaded with a sequential stream of scalar operations	The instructions are loaded with a sequential stream of multiple operations
Loading and scheduling instructions	The loaded instructions are dynamically scheduled by the hardware	The loaded statements are statically ordered by the compiler
Width of issue	The number of instructions loaded in parallel is fixed dynamically by the hardware	The number of instructions loaded in parallel is fixed statically by the compiler and the architecture
Order execution instructions	Dynamic loading allows execution in order and out of order	Static scheduling only allows one execution in order
Architectural Implications	The superscalar is a micro-architectural technique	VLIW is an architectural technique since hardware details are visible to the compiler / programmer.

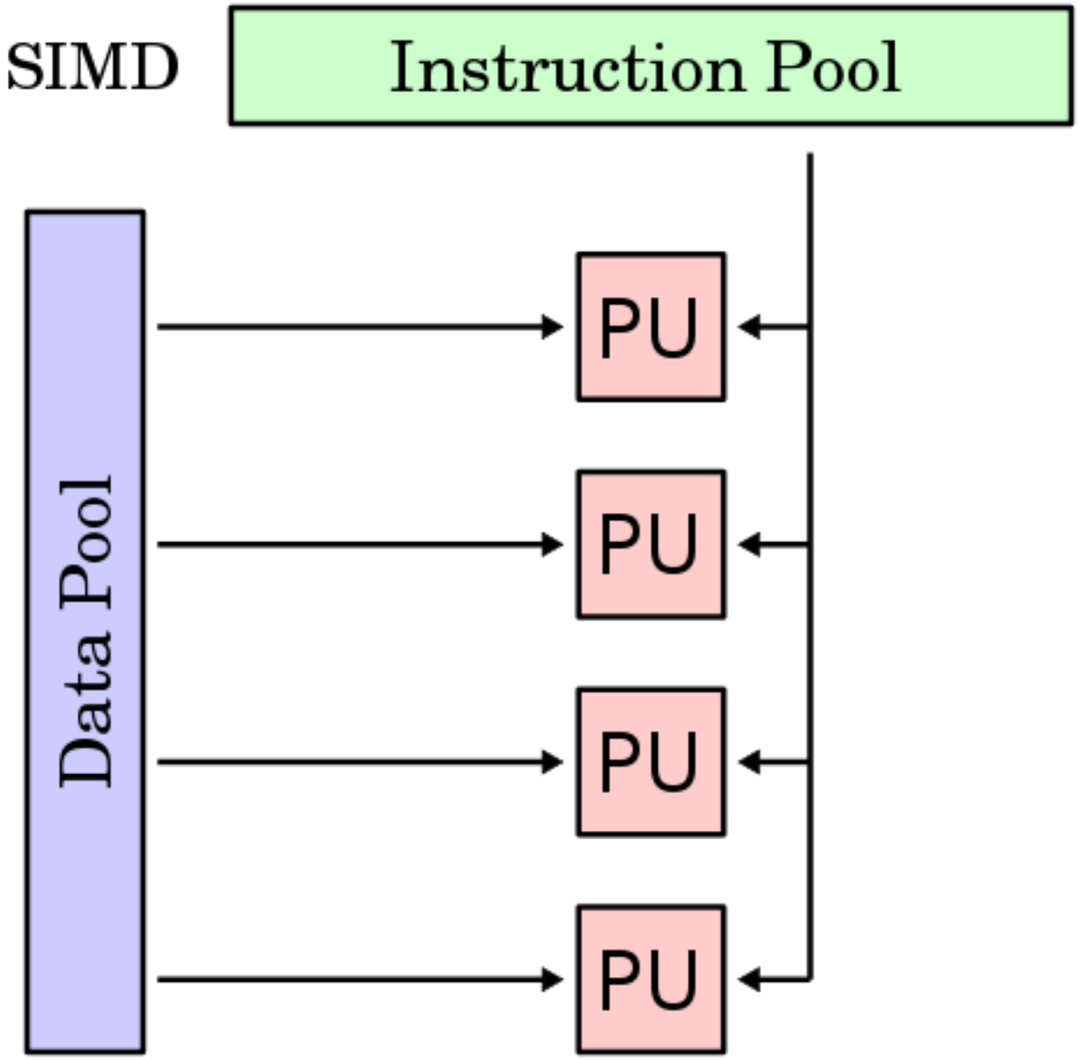
Source : Josh Fisher

Vector Extension

- Optimization possibilities
  - Duplication of functional units
    - exploited by superscalar and out-of-order
  - Expansion of functional units
    - towards SIMD programming
- Flynn's taxonomy
  - *SISD*: Single Instruction Single Data
  - *MISD*: Multiple Instruction Single Data
  - *SIMD*: Single Instruction Multiple Data
  - *MIMD*: Multiple Instruction Multiple Data

SIMD Instructions

- Idea is to use a single instruction (from the ISA) that will operate on multiple data at the same time
- Where is the data?
- In addition to scalar operands, the registers are extended to contain multiple scalar data (call a *vector*)
  - Vector registers



Advantages of SIMD

- Expands calculation performance at lower cost
  - More transistors to build these units
  - Depends on the operation
    - vector addition is like a simple addition without propagation
  - Chaining between vector operations is possible
- Minimal changes to the rest of the processor
  - No expansion of the pipeline as for a superscalar
  - No need to discover the parallelism of data
    - done by the compiler

Disadvantages of SIMD

- Exhibits optimization at the level of the architecture
  - Instruction set must provide multiple operations on data
  - Vector instructions support certain vector size
    - SSE (128 bits), AVX (256 bits), MIC/AVX512 (512 bits)
  - Need to generate specific code
- Imposes a certain degree of precision and parallelism
  - Example: 16 single-precision floats on MIC
- Applies to particular code classes
  - Problems with control flow
  - Regular scientific codes
  - Cryptography
  - Any code where simultaneous operations on multiple values of the same type are present and not difficult to discover

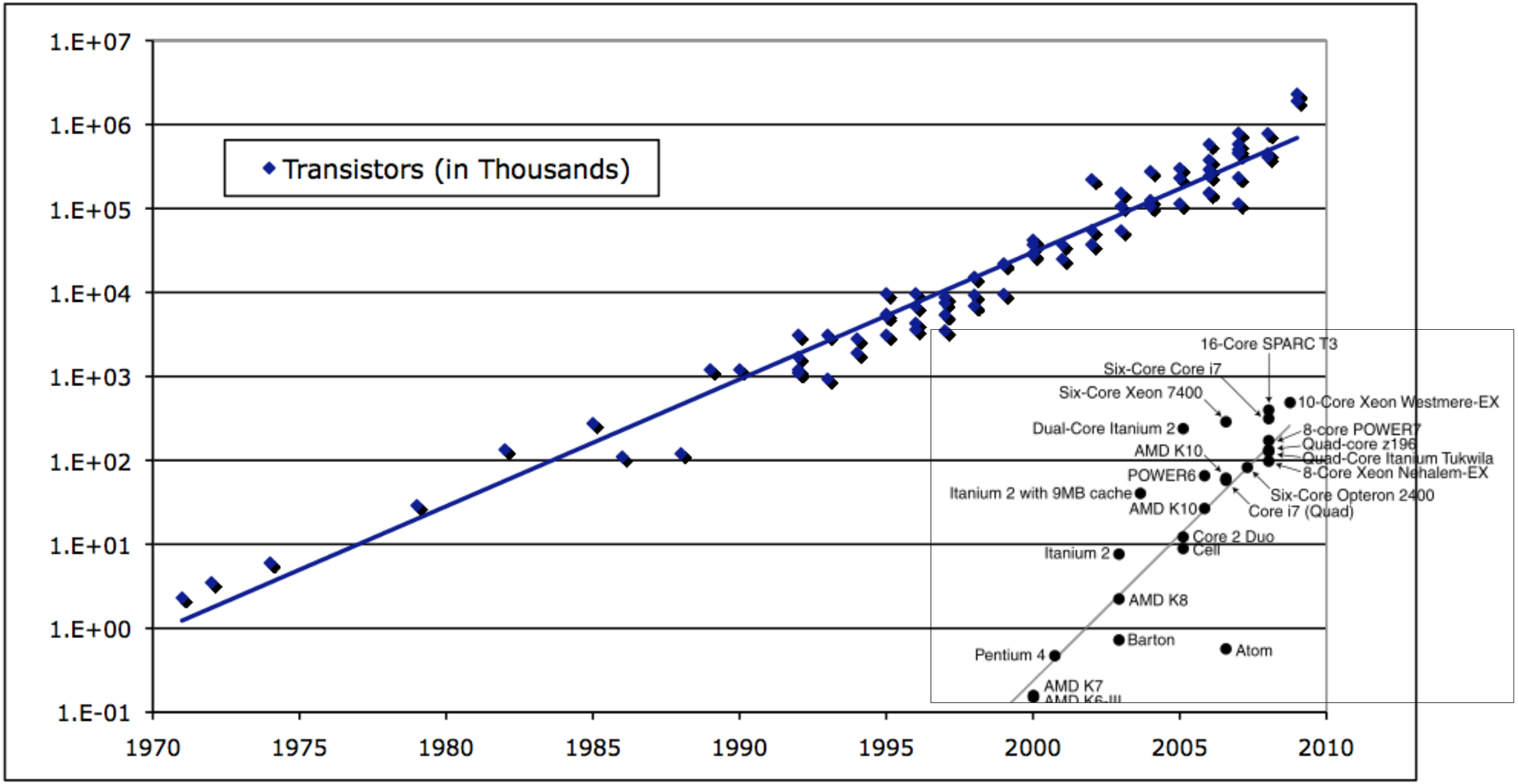
Conclusion from CM-1

- Steps to optimize a calculation core
  - Concept of pipeline
  - Mechanisms to avoid branches
    - out-of-order execution
    - branch prediction
  - Execute multiple instructions at a time
    - superscalar
    - VLIW
  - Multiple calculations per instruction
    - SIMD vector extension

Leveraging Moore's Law

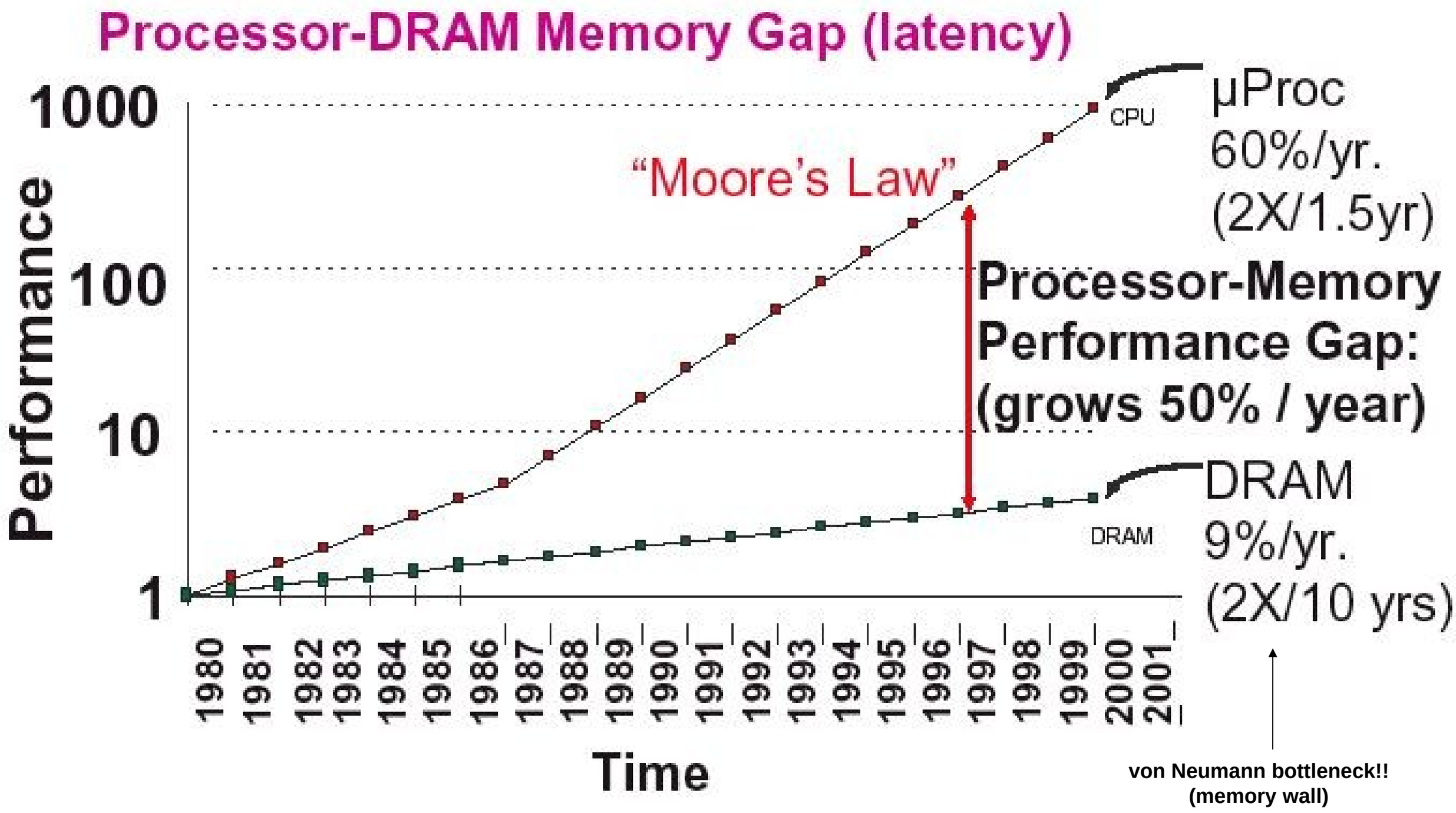
- Moore's Law (Gordon E. Moore, Intel co-founder)
  - # transistors in an integrated circuit doubles every 2 years
  - Observation or conjecture, not a physical or natural law
  - End of Moore's Law?
    - expected to continue to ~2020
- More transistors = more parallelism opportunities
- Microprocessors
  - Implicit parallelism
    - pipelining
    - multiple functional units
    - superscalar
  - Explicit parallelism
    - SIMD instructions
    - long instruction works
- Dennard scaling
  - Power requirements are proportional to area
  - Performance per watt grows with transistor density
  - Dennard scaling has broken down since 2007!

Microprocessor Transistor Counts (1971-2011)



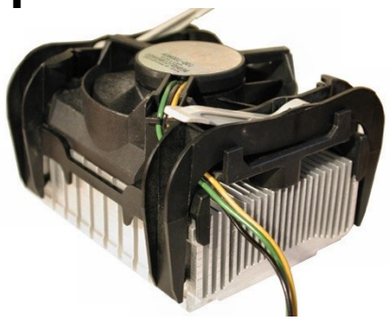
Data from Kunle Olukotun, Lance Hammond, Herb Sutter, Burton Smith, Chris Batten, and Krste Asanoviç
Slide from Kathy Yelick

What's Driving Computing Architecture?



What has happened in the last 15 years?

- Processing chip manufacturers increased processor performance by increasing CPU clock frequency
  - Riding Moore's law
- Until the chips got too hot!
  - Greater clock frequency greater electrical power
  - Breakdown of Dennard scaling
  - Pentium 4 heat sink
  - Frying an egg on a Pentium 4
- Add multiple cores to add performance
  - Keep clock frequency same or reduced
  - Keep lid on power requirements



Power Density Growth

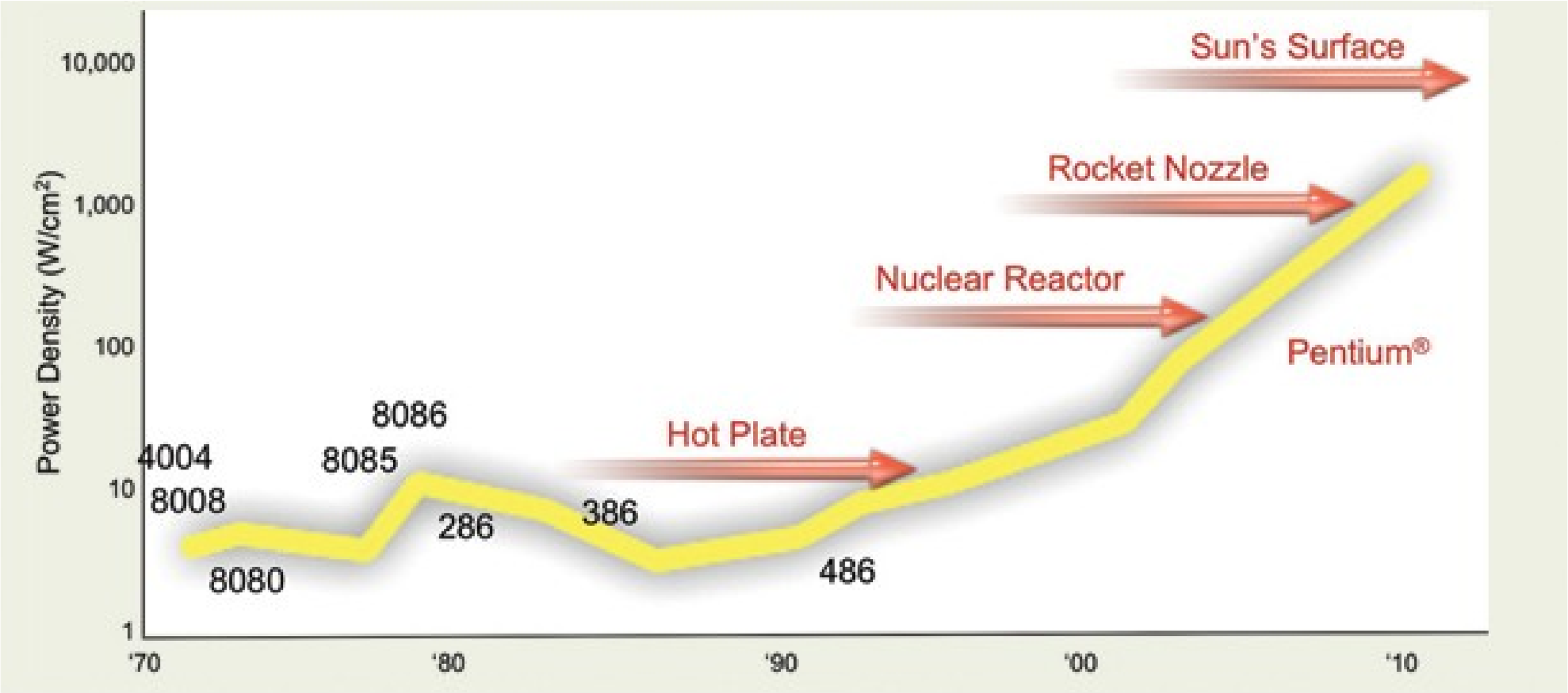
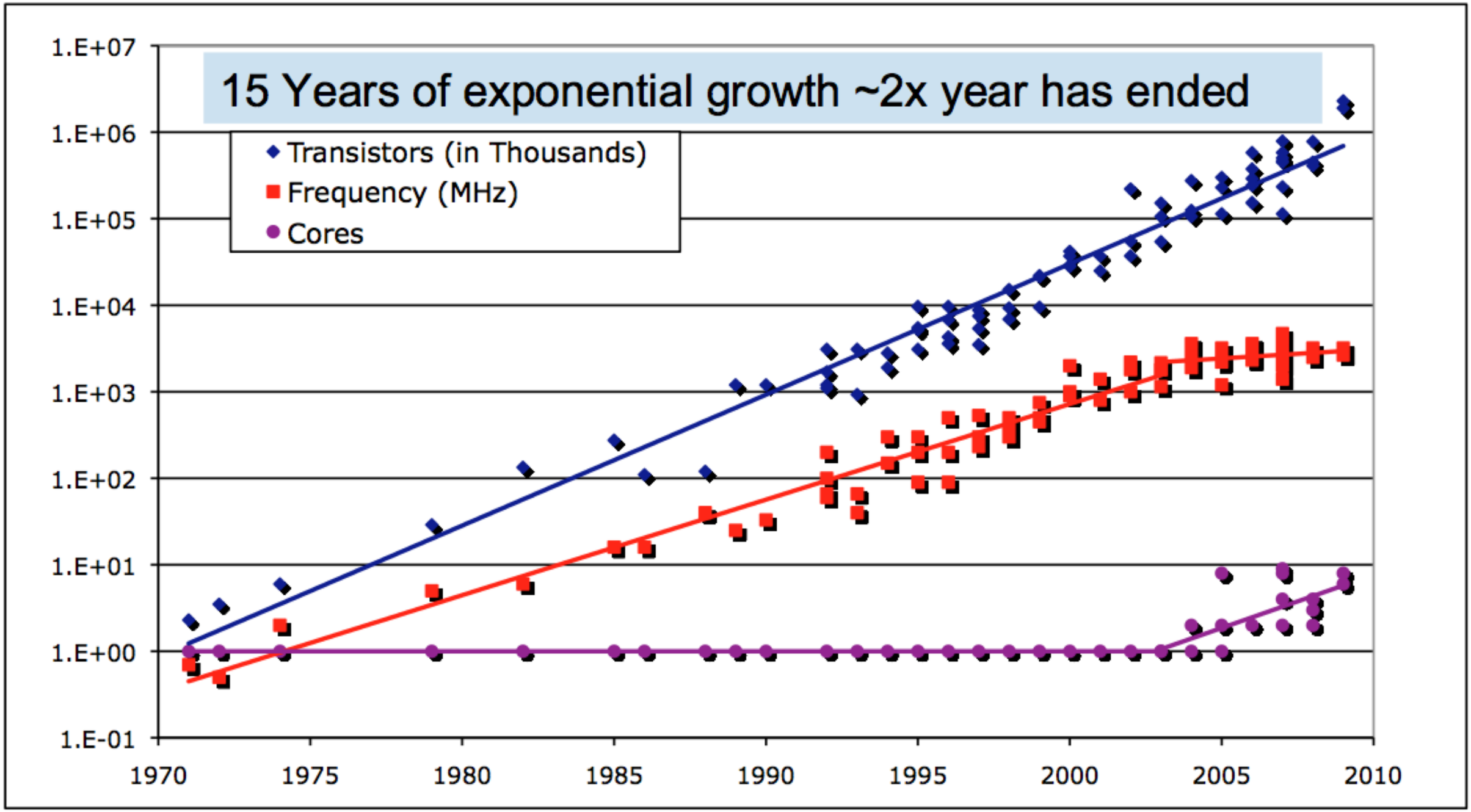


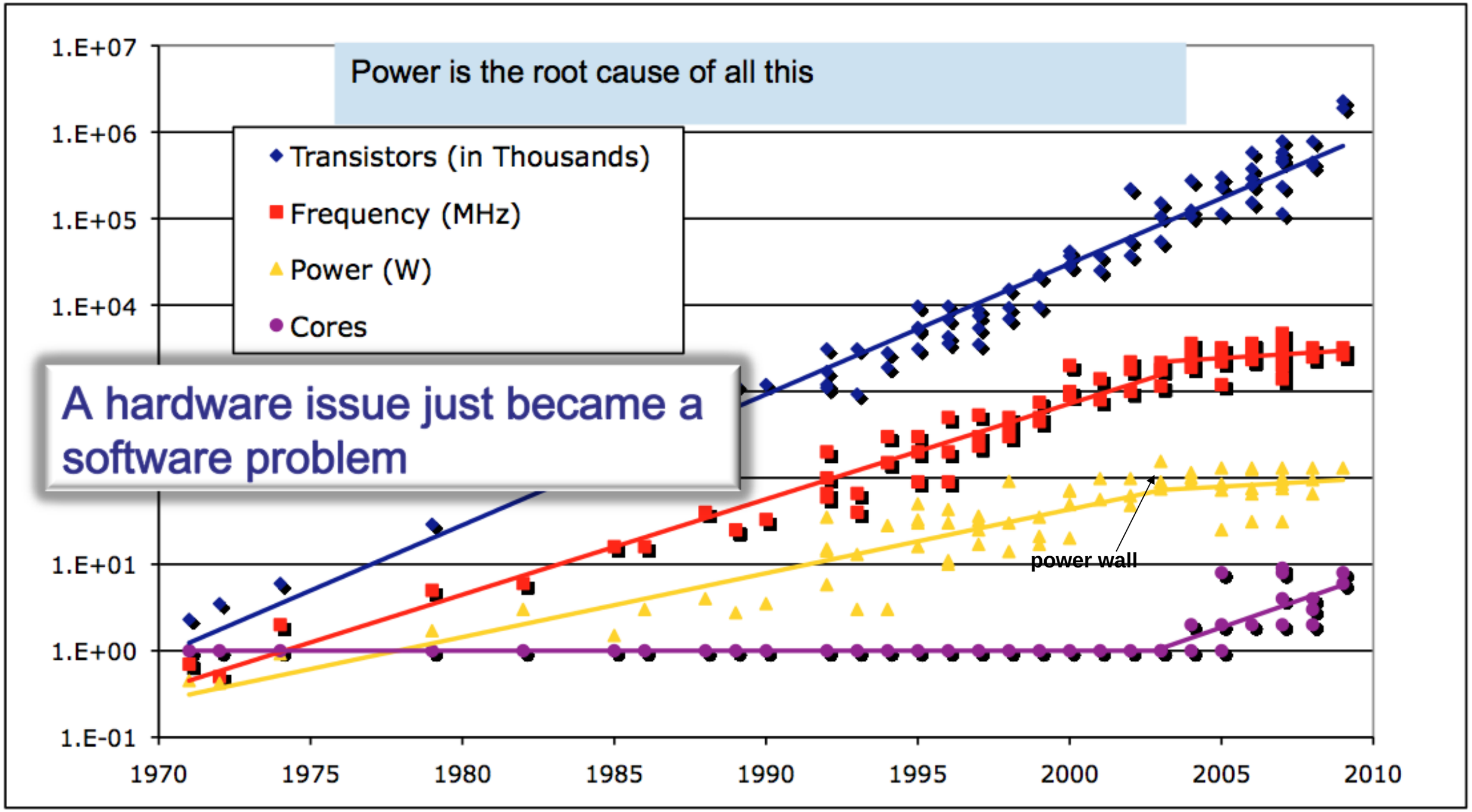
Figure courtesy of Pat Gelsinger, Intel Developer Forum, Spring 2004

What's Driving Parallel Computing Architecture?



Data from K. Olukotun, L. Hammand, H. Sutter, B. Smith, C. Batten, K. Asanovic; Slide from Kathy Yelick

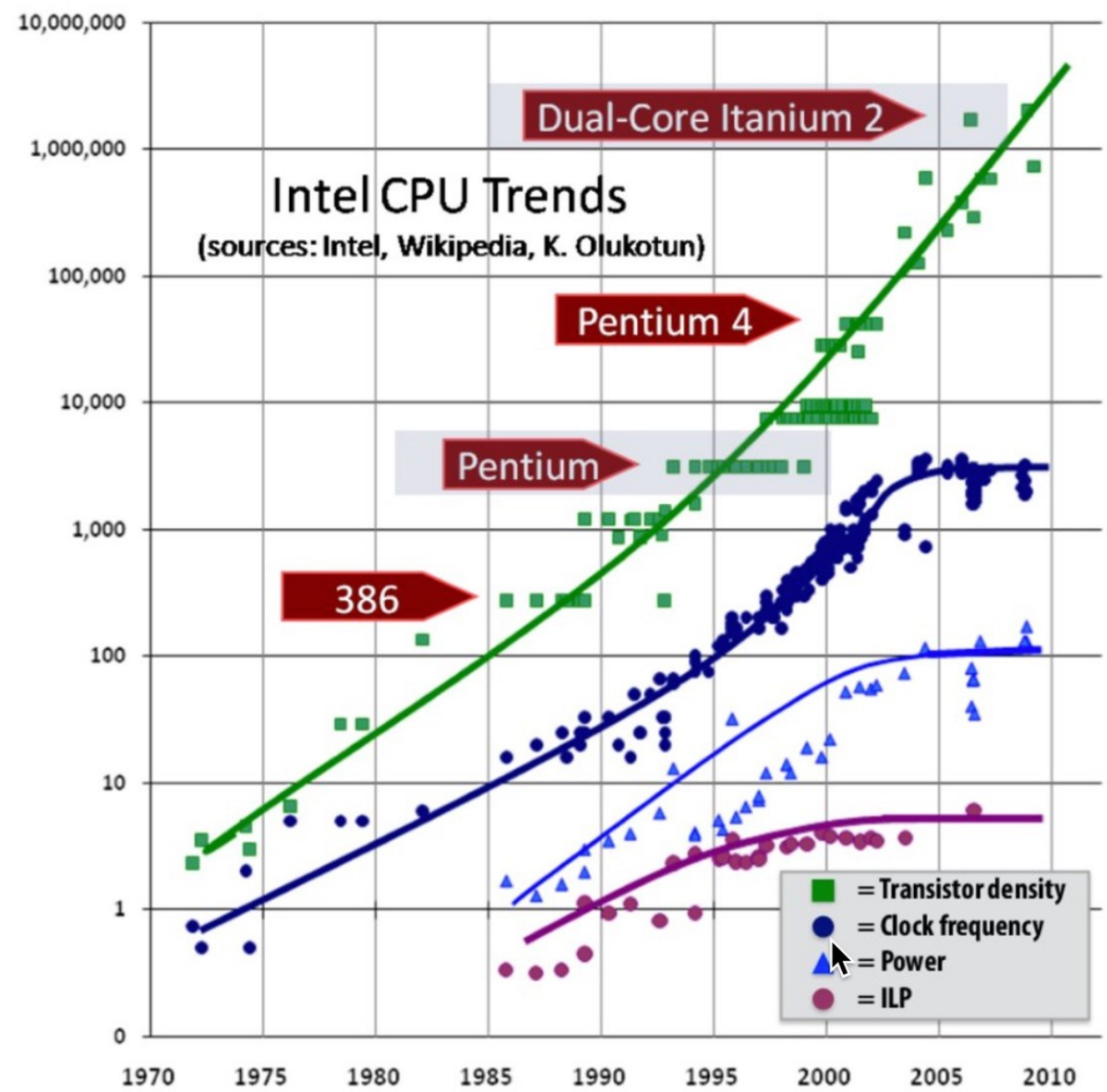
What's Driving Parallel Computing Architecture?



Data from K. Olukotun, L. Hammand, H. Sutter, B. Smith, C. Batten, K. Asanovic; Slide from Kathy Yelick

Single-core Performance Scaling

- Rate of single-instruction stream performance scaling has decreased to almost 0
- Frequency scaling is limited by power
- ILP scaling is tapped out
- CPU architects are now building faster processors by adding more execution units to run in parallel
- Parallel software must be written to see performance gains

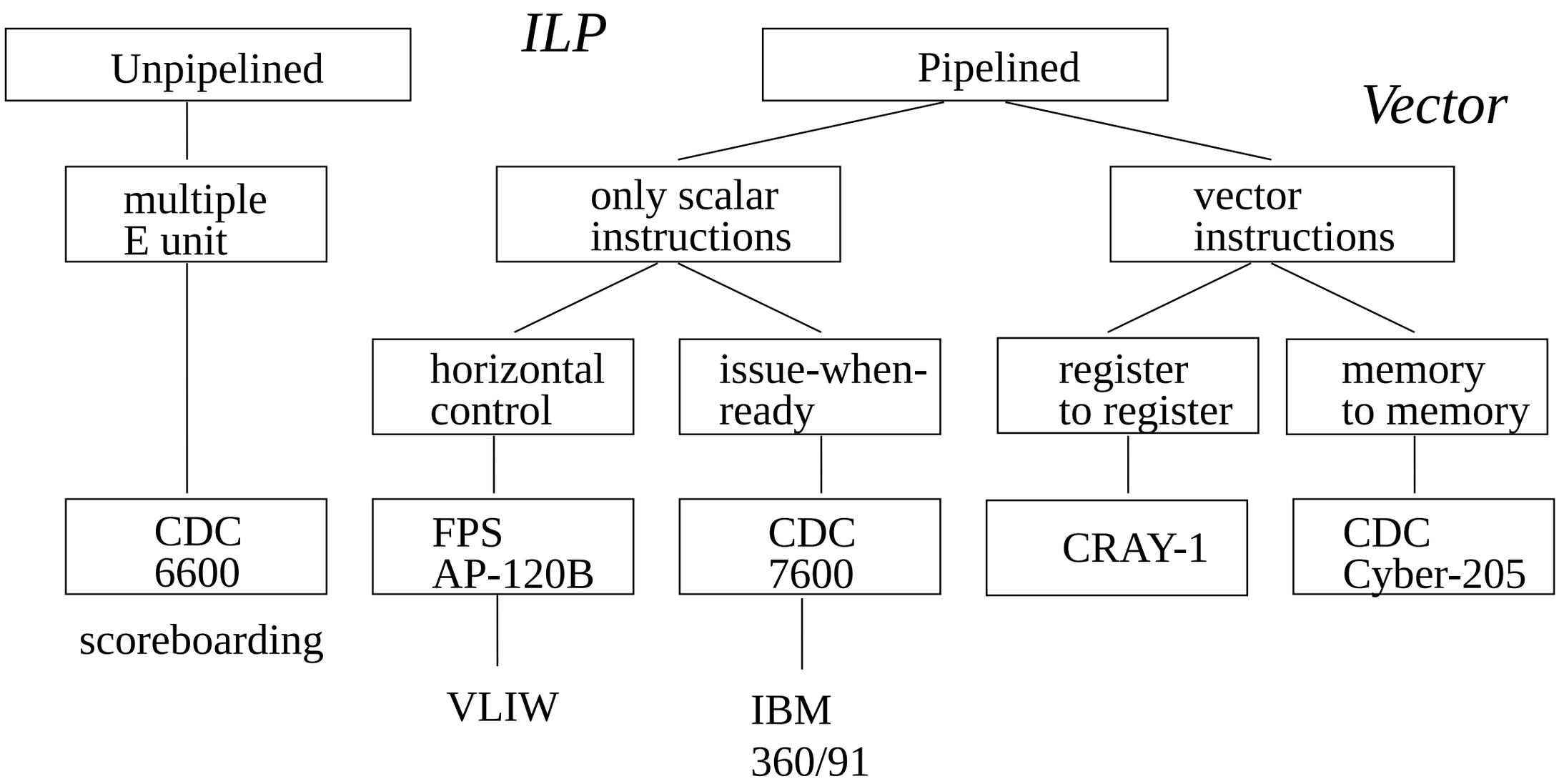


Phases of Supercomputing (Parallel) Architecture

- Phase 1 (1950s): sequential instruction execution
- Phase 2 (1960s): sequential instruction issue
  - Pipeline execution, reservations stations
  - Instruction Level Parallelism (ILP)
- Phase 3 (1970s): vector processors
  - Pipelined arithmetic units
  - Registers, multi-bank (parallel) memory systems
- Phase 4 (1980s): SIMD and SMPs
- Phase 5 (1990s): MPPs and clusters
  - Communicating sequential processors
- Phase 6 (2000s): many cores, accelerators, scale, …
- Phase 7 (2010s): many more cores, heterogeneity, …

Parallelism in Single Processor Computers

- History of processor architecture innovation



Vector Processing

- Scalar processing
  - Processor instructions operate on scalar values
  - Integer registers and floating point registers
- Vectors
  - Set of scalar data
  - Vector registers
    - integer, floating point (typically)
  - Vector instructions operate on vector registers (SIMD)
- Vector unit pipelining
- Multiple vector units
- Vector chaining

Liquid-cooled with inert fluorocarbon. (That's a fluorocarbon fountain!!!)

Cray 2