Εισαγωγή

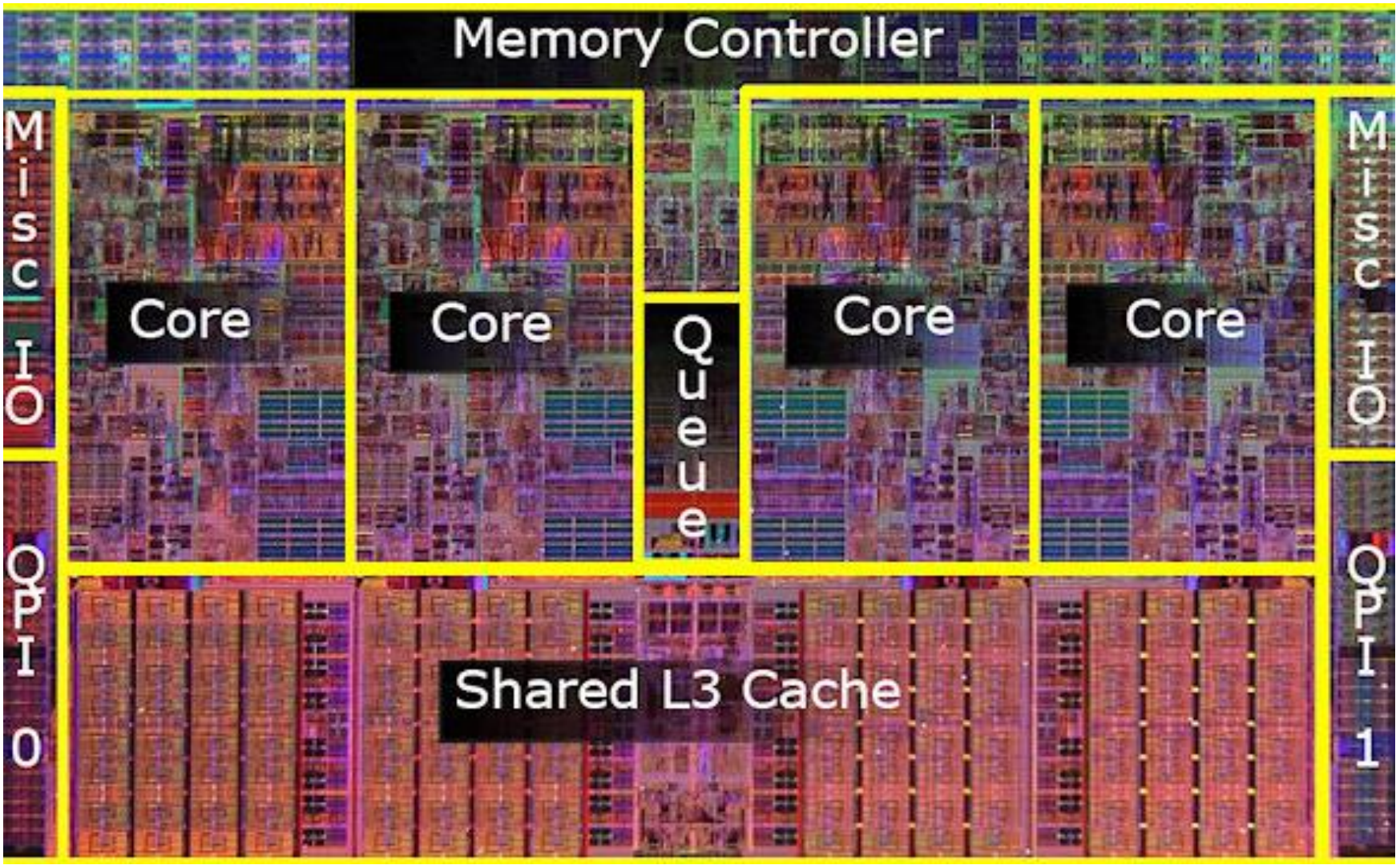
Σύνοψη βασικών εννοιών, 5-stage pipeline, επεκτάσεις για λειτουργίες πολλαπλών κύκλων

Νεκτάριος Κοζύρης & Διονύσης Πνευματικάτος
{nkoziris,pnevmati}@cslab.ece.ntua.gr

8ο εξάμηνο ΣΗΜΜΥ – Ακαδημαϊκό Έτος: 2019-20
http://www.cslab.ece.ntua.gr/courses/advcomparch/

Intel Core i7 Processor

Nahalem, 731M xtor, 263mm^2, 45nm process, ~2008

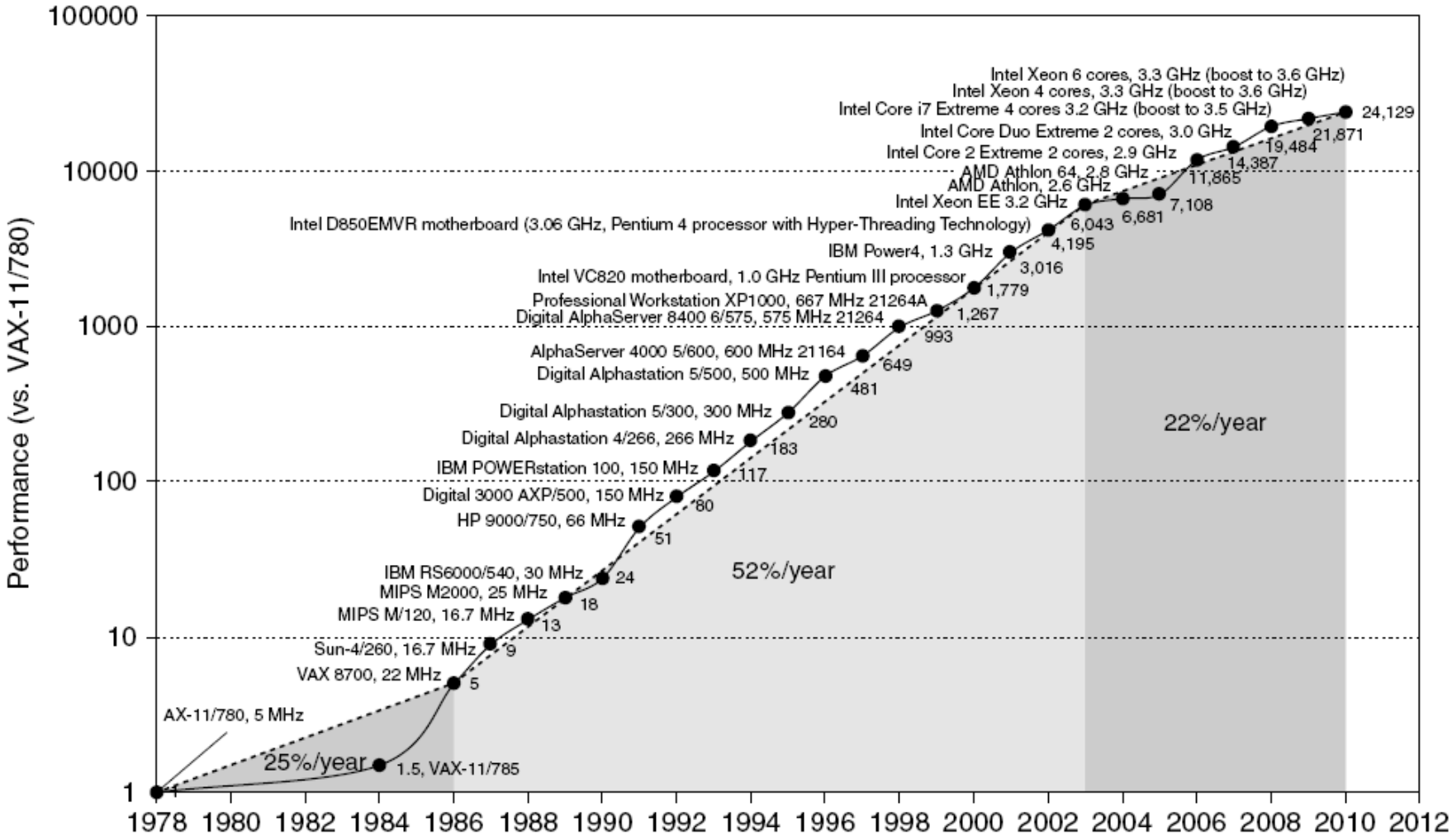


CPU die comparison

CPU	Technology Process	Cores	GPU	Transistor Count	Die Size mm^2
AMD Epyc Rome (64-bit, SIMD, caches)	7 & 12nm	32?		39.5B	1088(!)
Apple A13 (iphone11+)	7nm	4	GT2	8.5B	~99
Haswell GT3 4C	22nm	4	GT3	?	~264
Haswell GT2 4C	22nm	4	GT2	1.4B	177
Haswell ULT GT3 2C	22nm	2	GT3	1.3B	181
Intel Ivy Bridge 4C	22nm	4	GT2	1.2B	160
Intel Sandy Bridge E 6C	32nm	6	N/A	2.27B	435
Intel Sandy Bridge 4C	32nm	4	GT2	995M	216

Πηγές: cpu-world.com, anandtech.com, Wikipedia, κλπ.

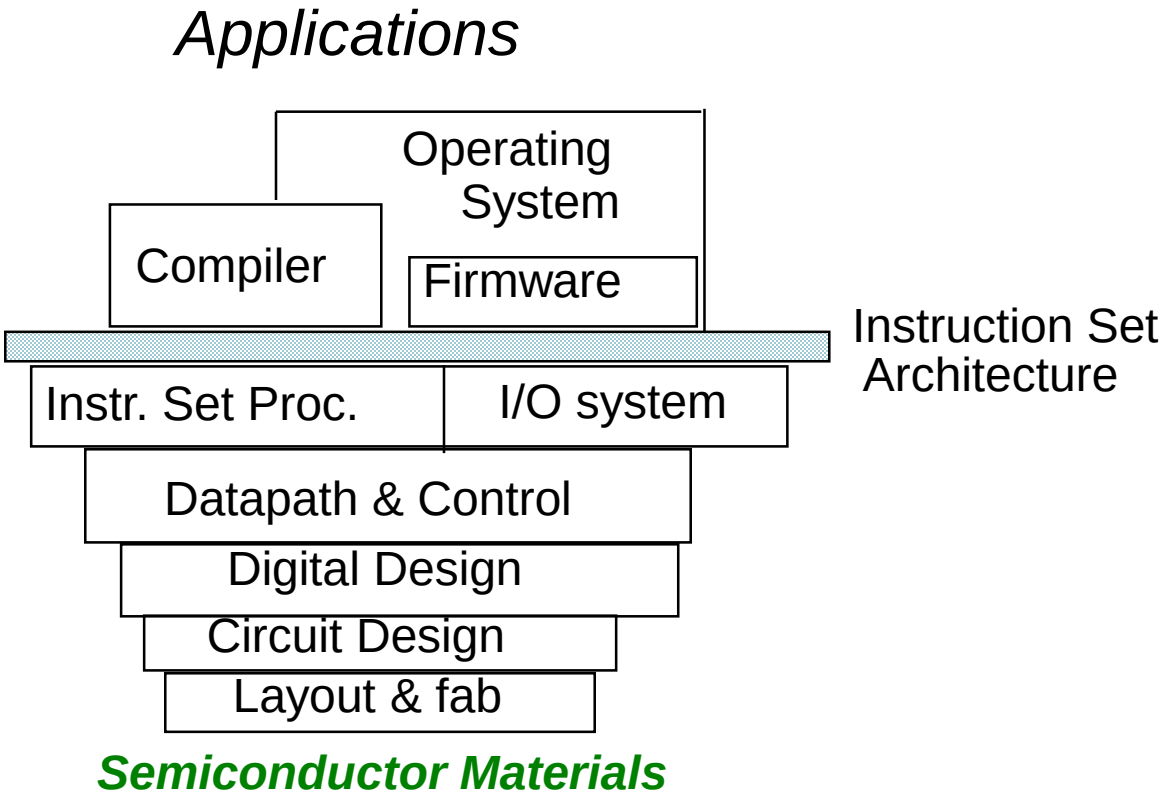
(Single) Processor Performance



Outline

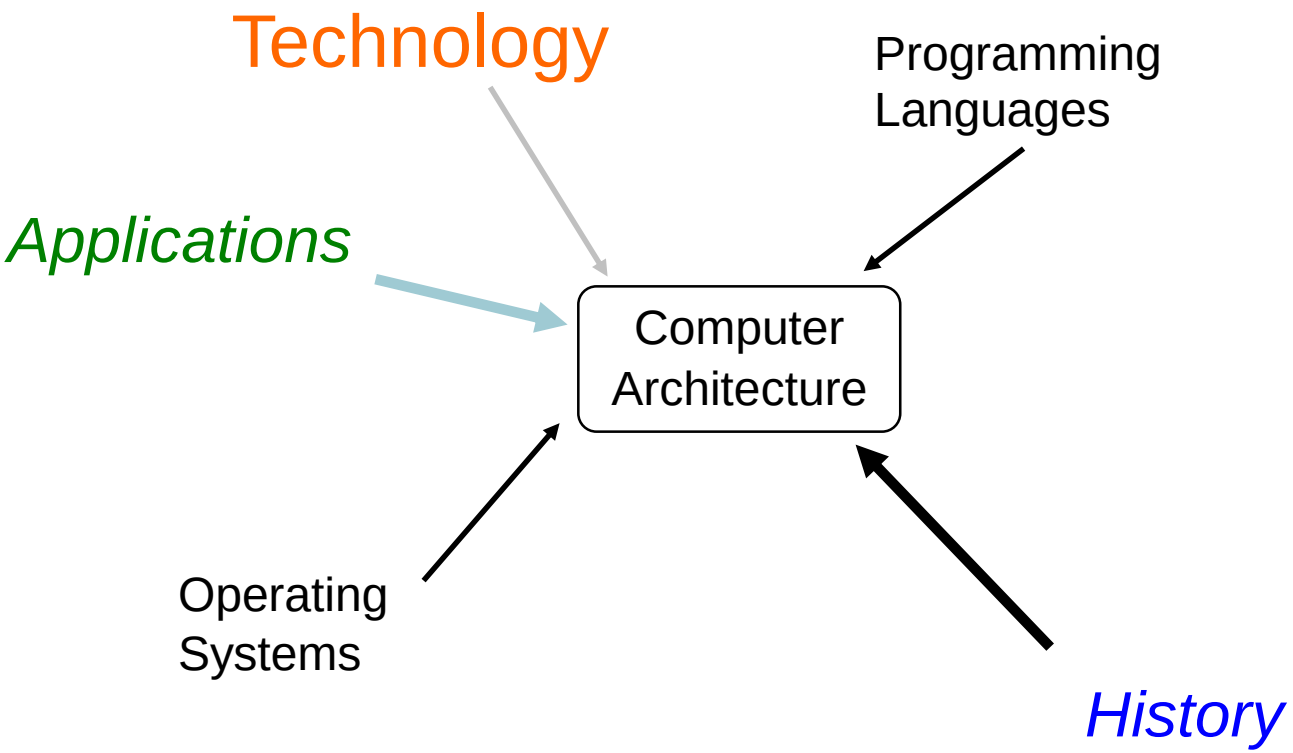
- What is Computer Architecture?
- Computer Instruction Sets – the fundamental abstraction
  - review and set up
- Dramatic Technology Advance
- Beneath the illusion – nothing is as it appears
- Computer Architecture Renaissance

What is “Computer Architecture”?

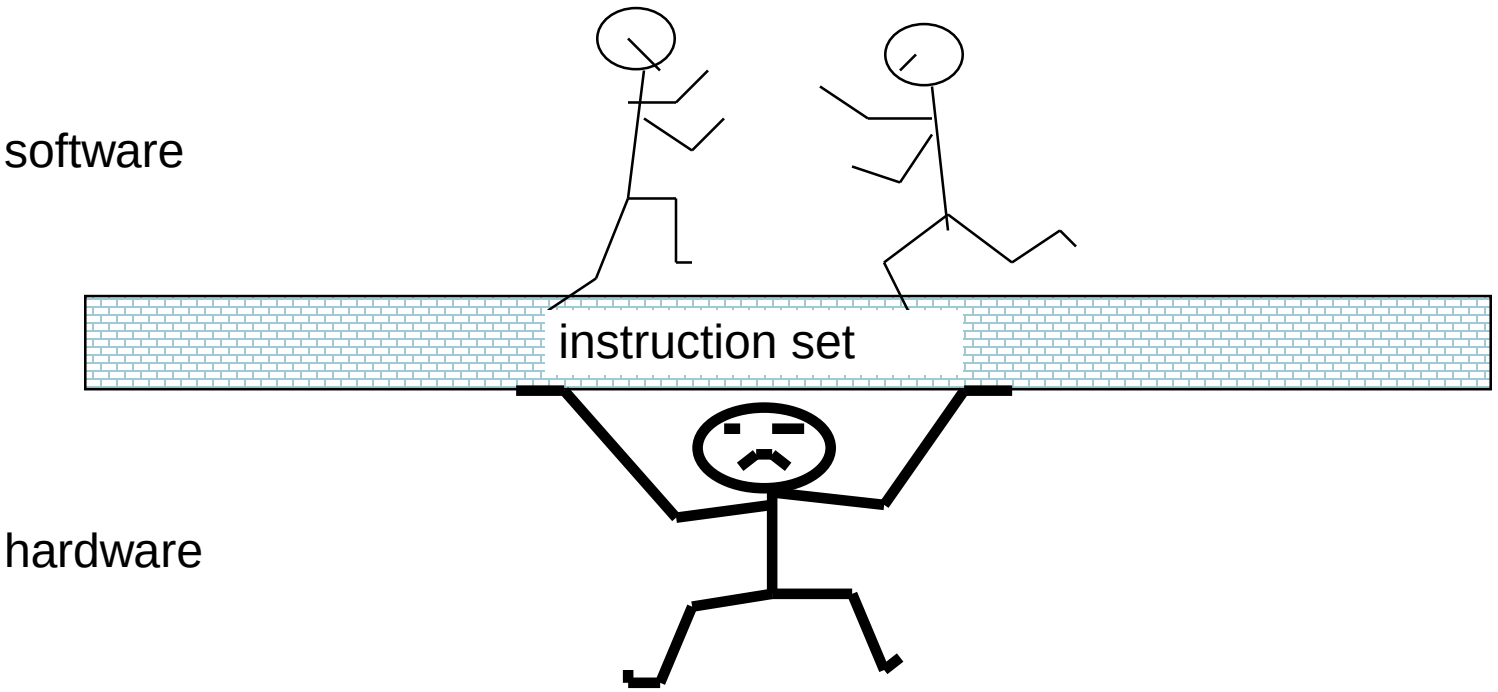


- Coordination of many *levels of abstraction*
- Under a rapidly changing set of forces
- Design, Measurement, *and* Evaluation

Forces on Computer Architecture

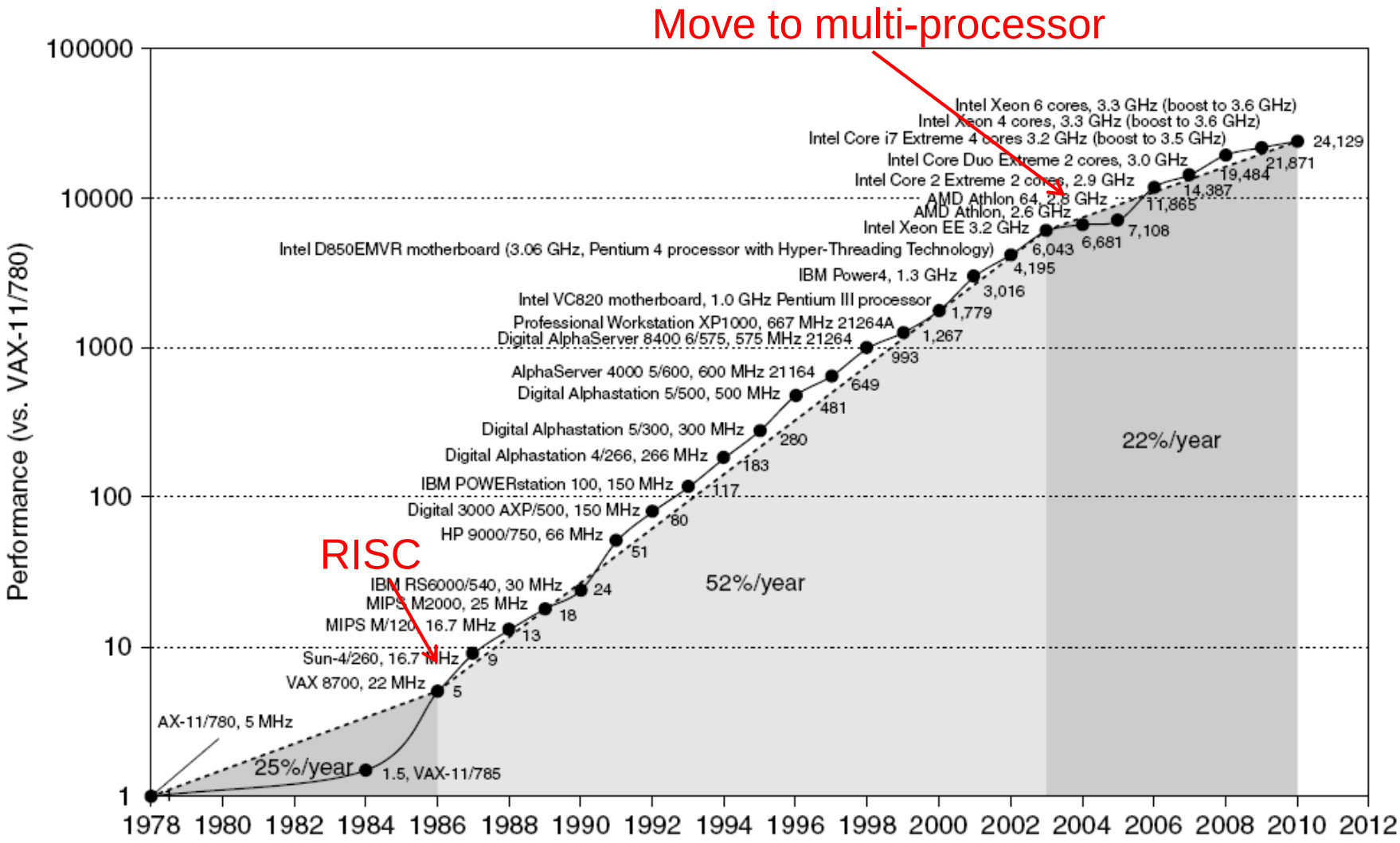


The Instruction Set: a Critical Interface



- Properties of a good abstraction
  - Lasts through many generations (portability)
  - Used in many different ways (generality)
  - Provides convenient functionality to higher levels
  - Permits an efficient implementation at lower levels

Single Processor Performance



Current Trends in Architecture

- Cannot continue to leverage Instruction-Level parallelism (ILP)
  - Single processor performance improvement ended in 2003

- New models for performance:
  - Data-level parallelism (DLP)
  - Thread-level parallelism (TLP)
  - Request-level parallelism (RLP)

- These require explicit restructuring of the application

Classes of Computers

- Personal Mobile Device (PMD)
  - e.g. smart phones, tablet computers (1.8 billion sold 2010)
  - Emphasis on energy efficiency and real-time
- Desktop Computing
  - Emphasis on price-performance (0.35 billion)
- Servers
  - Emphasis on availability (very costly downtime!), scalability, throughput (20 million)
- Clusters / Warehouse Scale Computers
  - Used for “Software as a Service (SaaS)”, PaaS, IaaS, etc.
  - Emphasis on availability ($6M/hour-downtime at Amazon.com!) and price-performance (power=80% of TCO!)
  - Sub-class: Supercomputers, emphasis: floating-point performance and fast internal networks, and big data analytics
- Embedded (IoT?) Computers (19 billion in 2010)
  - Emphasis: price

Re-Defining Computer Architecture

- "Old" view of computer architecture:
  - Instruction Set Architecture (ISA) design
  - i.e. decisions regarding:
    - Registers#, memory addressing, addressing modes, instruction operands, available operations, control flow instructions, instruction encoding
- "Real" computer architecture:
  - Specific requirements of the target machine
  - Design to maximize performance within constraints: cost, power, and availability
  - Includes ISA, microarchitecture, hardware
- Security? Quality of Service?

Trends in Technology

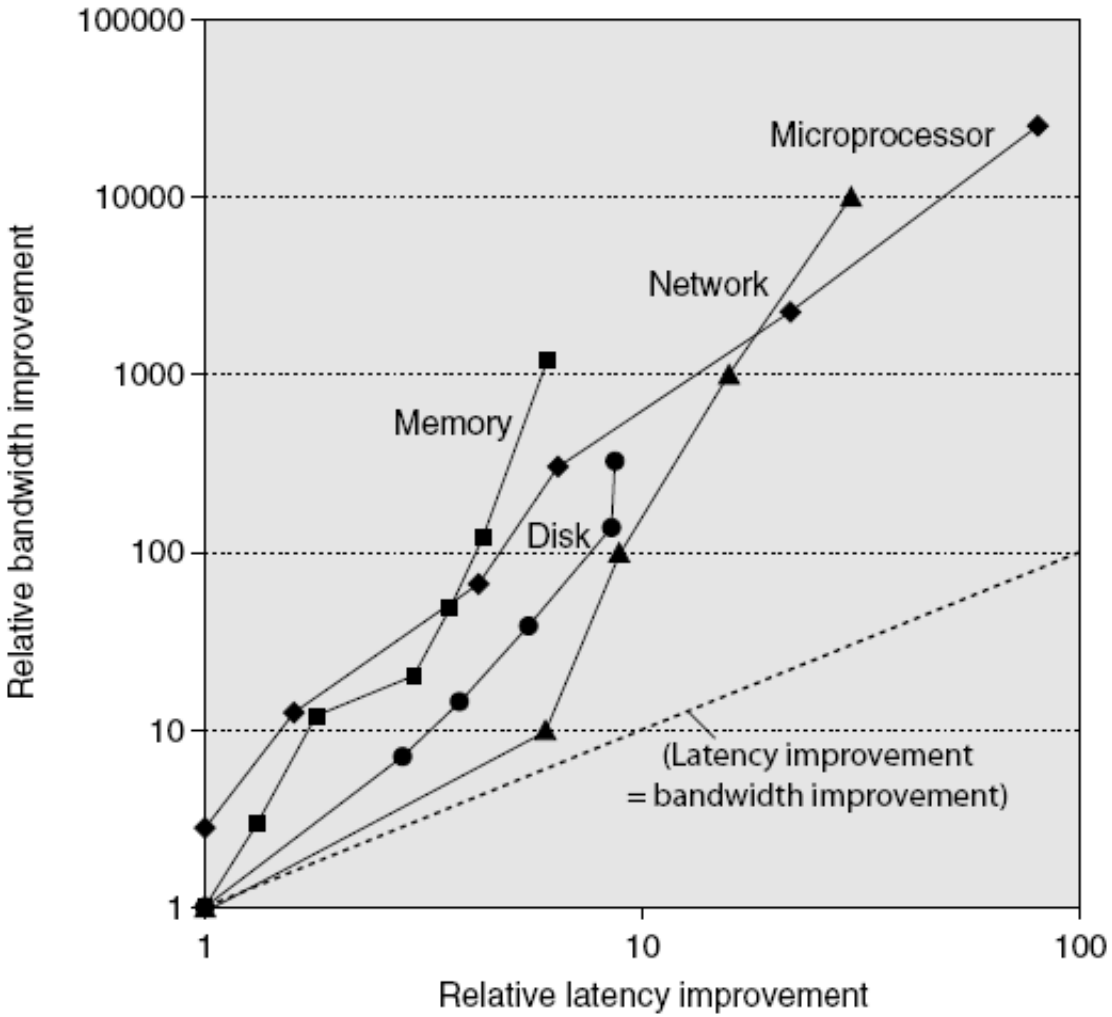
- Integrated circuit technology
  - Transistor density: 35%/year
  - Die size: 10-20%/year
  - Integration overall: 40-55%/year
- DRAM capacity: 25-40%/year (slowing)
- Flash capacity: 50-60%/year
  - 15-20X cheaper/bit than DRAM
- Magnetic disk technology: 40%/year
  - 15-25X cheaper/bit then Flash
  - 300-500X cheaper/bit than DRAM

Bandwidth and Latency

- Bandwidth or throughput
  - Total work done in a given time
  - 10,000-25,000X improvement for processors over the 1st milestone
  - 300-1200X improvement for memory and disks over the 1st milestone

- Latency or response time
  - Time between start and completion of an event
  - 30-80X improvement for processors over the 1st milestone
  - 6-8X improvement for memory and disks over the 1st milestone

Bandwidth and Latency



Log-log plot of bandwidth and latency milestones

Transistors and Wires

- Feature size
  - Minimum size of transistor or wire in x or y dimension
  - 10 microns in 1971 to .032 microns in 2011
  - Transistor performance scales linearly
    - Wire delay does not improve with feature size!
  - Integration density scales quadratically
  - Linear performance and quadratic density growth present a challenge and opportunity, creating the need for computer architect!

Power and Energy

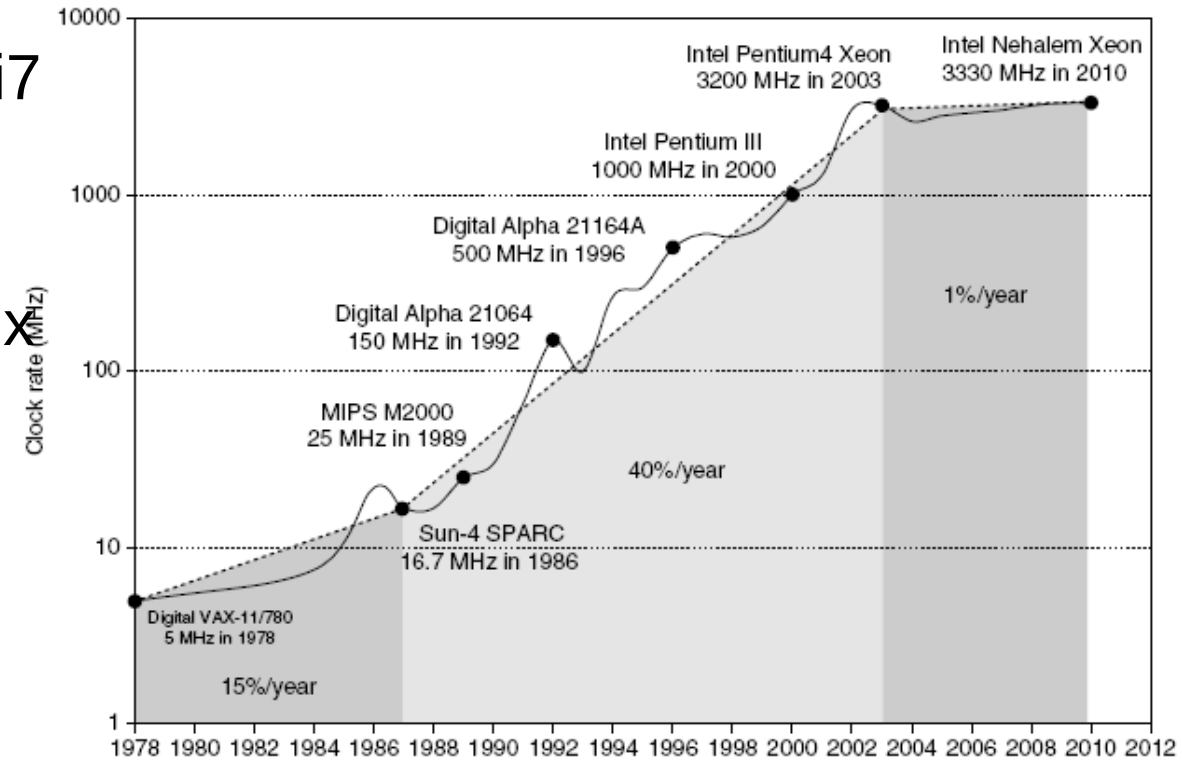
- Problem: Get power in, get power out
- Thermal Design Power (TDP)
  - Characterizes sustained power consumption
  - Used as target for power supply and cooling system
  - Lower than peak power, higher than average power consumption
  - Envelop?
- Clock rate can be reduced dynamically to limit power consumption
- Energy per task is often a better measurement

Dynamic Energy and Power

- Dynamic energy
  - Transistor switch from 0 -> 1 or 1 -> 0
  - f x Capacitive load x Voltage^2
    - f is activity factor
  - For f= ½ we get ½ x Capacitive load x Voltage^2
    - Typical assumption for activity factor is ½
- Dynamic power
  - ½ x Capacitive load x Voltage2 x Frequency switched
    - Again, assumes activity factor is ½
- Reducing clock rate reduces power, not energy

Power

- Intel 80386 consumed ~ 2 W
- 3.3 GHz Intel Core i7 consumes 130 W
- Heat must be dissipated from 1.5 x 1.5 cm chip
- This is the limit of what can be cooled by air



Reducing Power

- Techniques for reducing power:
  - Do nothing
  - Dynamic Voltage Scaling (DVS)
  - Dynamic Frequency Scaling (DFS)
  - Dynamic Voltage-Frequency Scaling (DVFS)
  - Low power state for DRAM, disks
    - Sleep modes
  - Overclocking, then turn off cores (!!!)

Static Power

- Static power consumption
  - Currentstatic x Voltage
  - Scales with number of transistors
  - To reduce: power gating
  - Race-to-halt
- The new primary evaluation for design innovation
  - Tasks per joule
  - Performance per watt

Trends in Cost

- Cost driven down by learning curve
  - Yield
- DRAM: price closely tracks cost
- Microprocessors: price depends on volume
  - 10% less for each doubling of volume

Integrated Circuit Cost

- Integrated circuit

$$\text{Cost of integrated circuit} = \frac{\text{Cost of die} + \text{Cost of testing die} + \text{Cost of packaging and final test}}{\text{Final test yield}}$$

$$\text{Cost of die} = \frac{\text{Cost of wafer}}{\text{Dies per wafer} \times \text{Die yield}}$$

$$\text{Dies per wafer} = \frac{\pi \times (\text{Wafer diameter}/2)^2}{\text{Die area}} - \frac{\pi \times \text{Wafer diameter}}{\sqrt{2 \times \text{Die area}}}$$

$$\text{Die yield} = \text{Wafer yield} \times 1/(1 + \text{Defects per unit area} \times \text{Die area})^N$$

- Bose-Einstein formula (!)
- Defects per unit area = 0.016-0.057 defects per square cm (2010)
- N = process-complexity factor = 11.5-15.5 (40 nm, 2010)
- *The manufacturing process dictates the wafer cost, wafer yield and defects per unit area*
- *The architect's design affects the die area, which in turn affects the defects and cost per die*

Dependability

- Systems alternate between two states of service with respect to Service Level Agreements/Objectives (SLA/SLO):
  - Service accomplishment, where service is delivered as specified by SLA
  - Service interruption, where the delivered service is different from the SLA
- Module reliability:
  - Mean time to failure (MTTF)
  - Mean time to repair (MTTR)
  - Mean time between failures (MTBF) = MTTF + MTTR
  - Availability = MTTF / MTBF

Measuring Performance

- Typical performance metrics:
  - Response time
  - Throughput
- Speedup of X relative to Y
  - Execution timeY / Execution timeX
- Execution time
  - Wall clock time: includes all system overheads
  - CPU time: only computation time
- Benchmarks
  - Kernels (e.g. matrix multiply)
  - Toy programs (e.g. sorting)
  - Synthetic benchmarks (e.g. Dhrystone)
  - Benchmark suites (e.g. SPEC06fp, TPC-C)

Principles of Computer Design

- Take Advantage of Parallelism
  - e.g. multiple processors, disks, memory banks, pipelining, multiple functional units
- Principle of Locality
  - Reuse of data and instructions
- Focus on the Common Case!!!
  - Amdahl's Law

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times \left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)$$

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

Amdahl's Law

$$\text{ExTime}_{new} = \text{ExTime}_{old} \times \left[\left(1 - \text{Fraction}_{enhanced}\right) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}} \right]$$

$$\text{Speedup}_{overall} = \frac{\text{ExTime}_{old}}{\text{ExTime}_{new}} = \frac{1}{\left(1 - \text{Fraction}_{enhanced}\right) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}}}$$

F=0.1 => S ~= 1.1
F=0.5 => S = 2
F=0.9 => S = 10

Best you could ever hope to get:

$$\text{Speedup}_{maximum} = \frac{1}{\left(1 - \text{Fraction}_{enhanced}\right)}$$

Principles of Computer Design

- The Processor Performance Equation

$$\text{CPU time} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

$$\text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}}$$

$$\text{CPU time} = \text{Instruction count} \times \text{Cycles per instruction} \times \text{Clock cycle time}$$

$$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}$$

Principles of Computer Design

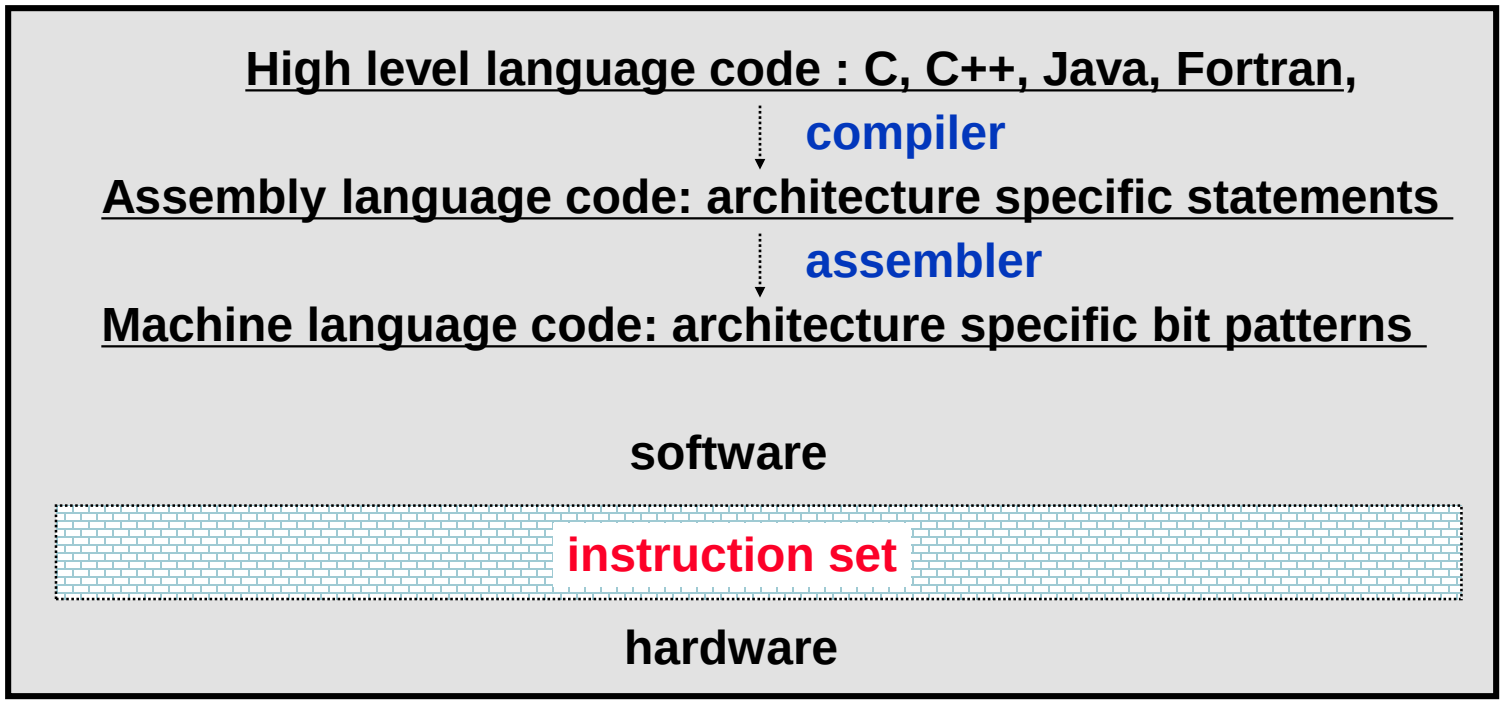
- Different instruction types having different CPIs

$$\text{CPU clock cycles} = \sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i$$

$$\text{CPU time} = \left(\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i \right) \times \text{Clock cycle time}$$

Instruction Set Architecture (ISA)

- Serves as an interface between software and hardware.
- Provides a mechanism by which the software tells the hardware what should be done.



Instruction Set Design Issues

- Instruction set design issues include:
  - Where are operands stored?
    - registers, memory, stack, accumulator
  - How many explicit operands are there?
    - 0, 1, 2, or 3
  - How is the operand location specified?
    - register, immediate, indirect, . . .
  - What type & size of operands are supported?
    - byte, int, float, double, string, vector. . .
  - What operations are supported?
    - add, sub, mul, move, compare . . .

Classifying ISAs

Accumulator (before 1960, e.g. 68HC11):

1-address	add A	acc ← acc + mem[A]

Stack (1960s to 1970s):

0-address	add	tos ← tos + next

Memory-Memory (1970s to 1980s):

2-address	add A, B	mem[A] ← mem[A] + mem[B]
3-address	add A, B, C	mem[A] ← mem[B] + mem[C]

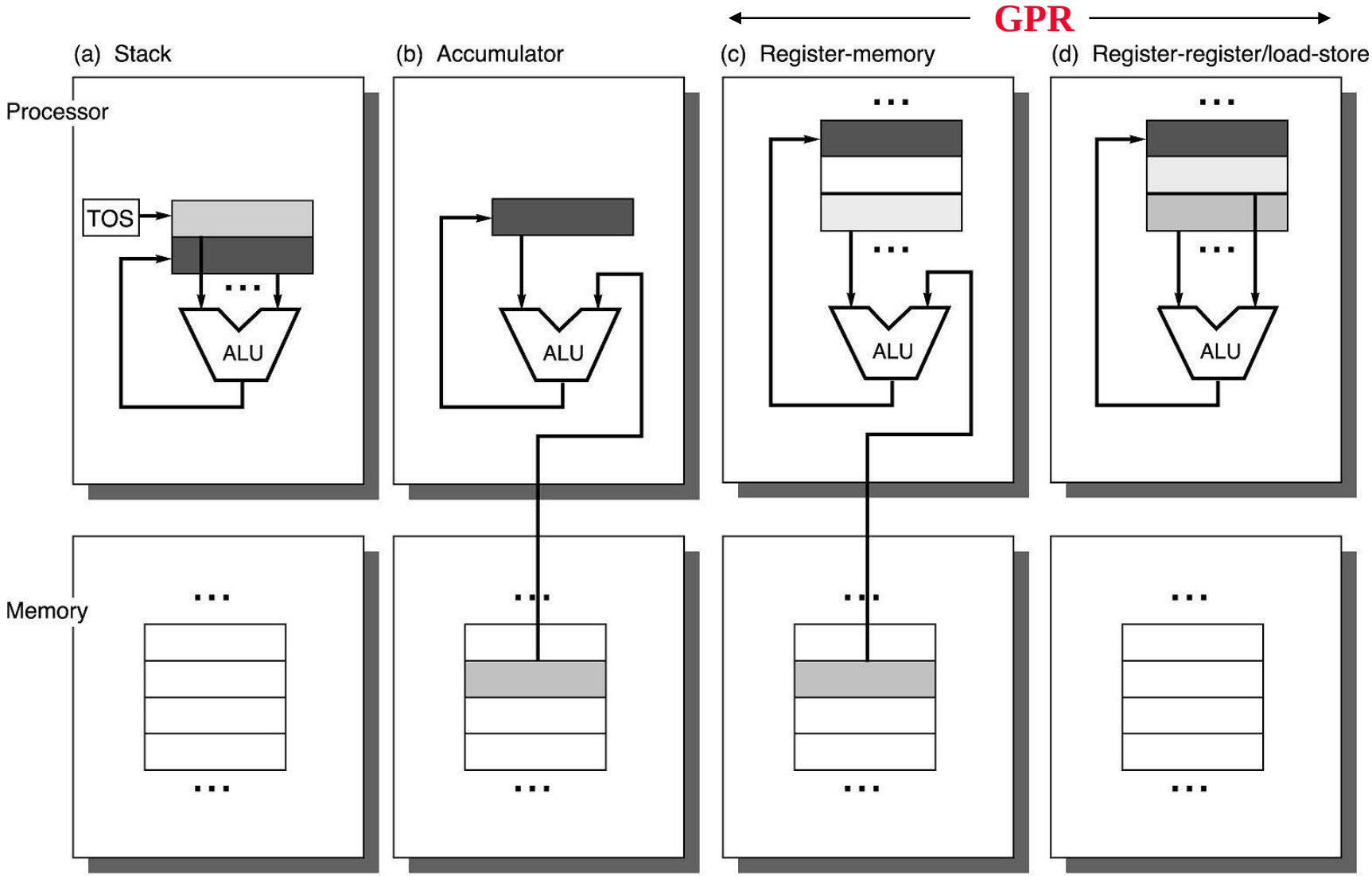
Register-Memory (1970s to present, e.g. 80x86):

2-address	add R1, A	R1 ← R1 + mem[A]
	load R1, A	R1 ← mem[A]

Register-Register (Load/Store/RISC) ('60s to present, e.g. MIPS):

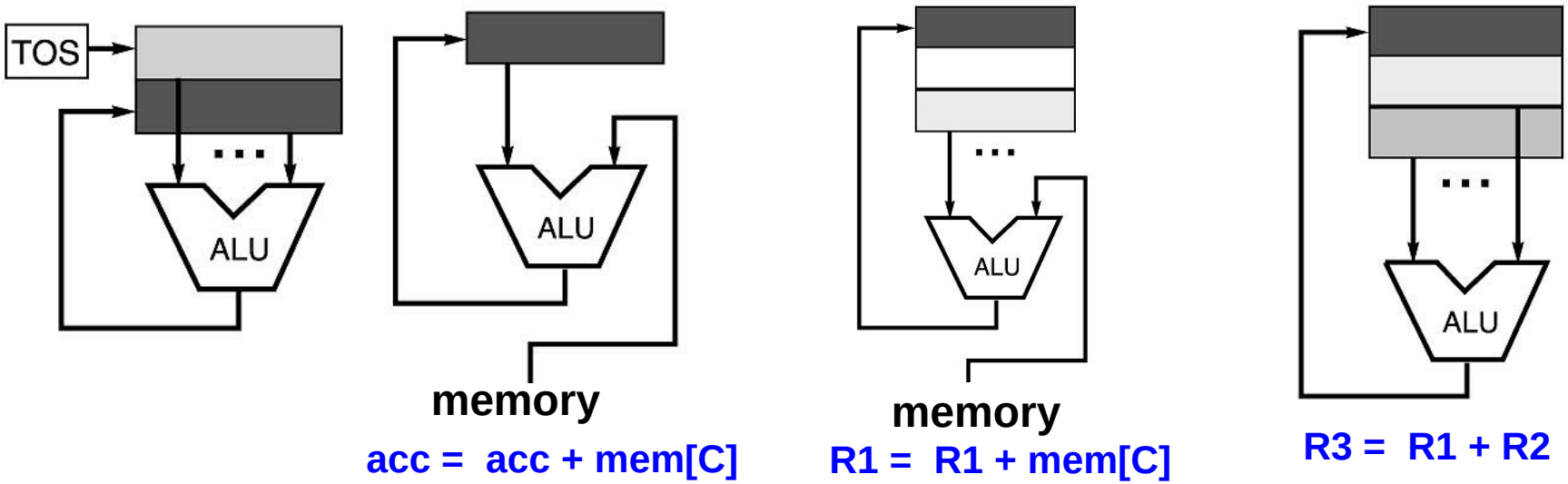
3-address	add R1, R2, R3	R1 ← R2 + R3
	load R1, R2	R1 ← mem[R2]
	store R1, R2	mem[R1] ← R2

Operand Locations in Four ISA Classes



Code Sequence C = A + B for 4 ISAs

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A Push B Add Pop C	Load A Add B Store C	Load R1, A Add R1, B Store C, R1	Load R1,A Load R2, B Add R3, R1, R2 Store C, R3



Types of Addressing Modes (VAX)

	Addressing Mode	Example	Action
1.	Register direct	Add R4, R3	R4 <- R4 + R3
2.	Immediate	Add R4, #3	R4 <- R4 + 3
3.	Displacement	Add R4, 100(R1)	R4 <- R4 + M[100 + R1]
4.	Register indirect	Add R4, (R1)	R4 <- R4 + M[R1]
5.	Indexed	Add R4, (R1 + R2)	R4 <- R4 + M[R1 + R2]
6.	Direct	Add R4, (1000)	R4 <- R4 + M[1000]
7.	Memory Indirect	Add R4, @(R3)	R4 <- R4 + M[M[R3]]
8.	Autoincrement	Add R4, (R2)+	R4 <- R4 + M[R2] R2 <- R2 + d
9.	Autodecrement	Add R4, (R2)-	R4 <- R4 + M[R2] R2 <- R2 - d
10.	Scaled	Add R4, 100(R2)[R3]	R4 <- R4 + M[100 + R2 + R3*d]

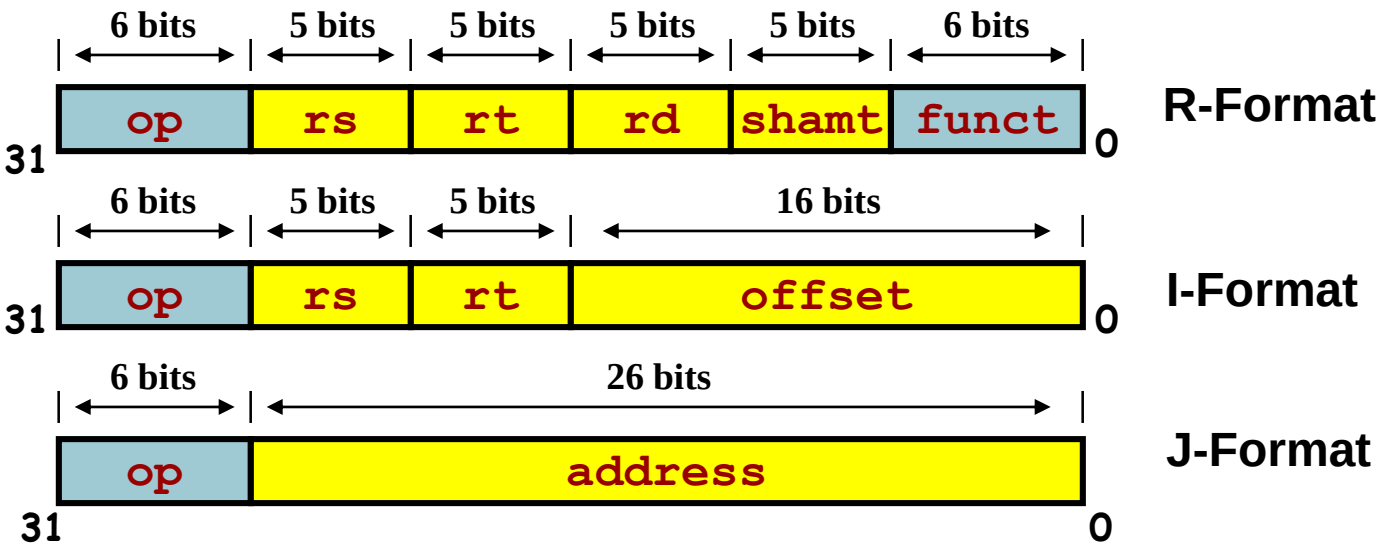
- [Clark and Emer]: modes 1-4 => ***93%*** of all operands on the VAX!!!

Types of Operations

- Arithmetic and Logic: AND, ADD
- Data Transfer: MOVE, LOAD, STORE
- Control BRANCH, JUMP, CALL
- System OS CALL, VM
- Floating Point ADDF, MULF, DIVF
- Decimal ADDD, CONVERT
- String MOVE, COMPARE
- Graphics (DE)COMPRESS

MIPS Instructions

- All instructions exactly 32 bits wide
- Different formats for different purposes
- Similarities in formats ease implementation



MIPS Instruction Types

- Arithmetic & Logical - manipulate data in registers

  add $s1, $s2, $s3 $s1 = $s2 + $s3

  or $s3, $s4, $s5 $s3 = $s4 OR $s5

- Data Transfer - move register data to/from memory → load & store

  lw $s1, 100($s2) $s1 = Memory[$s2 + 100]

  sw $s1, 100($s2) Memory[$s2 + 100] = $s1

- Branch - alter program flow

  beq $s1, $s2, 25 if ($s1==$s1) PC = PC + 4 + 4*25

  else PC = PC + 4

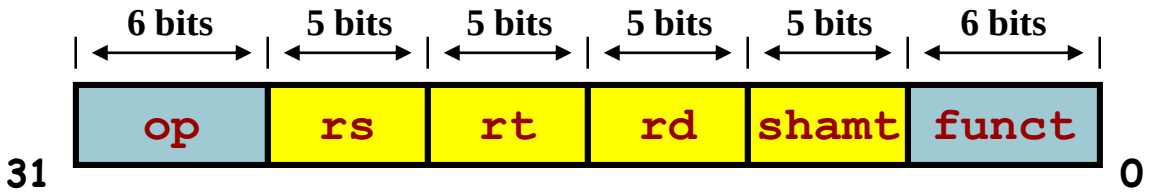
MIPS Arithmetic & Logical Instructions

- Instruction usage (assembly)

add dest, src1, src2	dest=src1 + src2
sub dest, src1, src2	dest=src1 - src2
and dest, src1, src2	dest=src1 AND src2

- Instruction characteristics
  - Always 3 operands: destination + 2 sources
  - Operand order is fixed
  - Operands are always general purpose registers
- Design Principles:
  - Design Principle 1: Simplicity favors regularity
  - Design Principle 2: Smaller is faster

Arithmetic & Logical Instructions



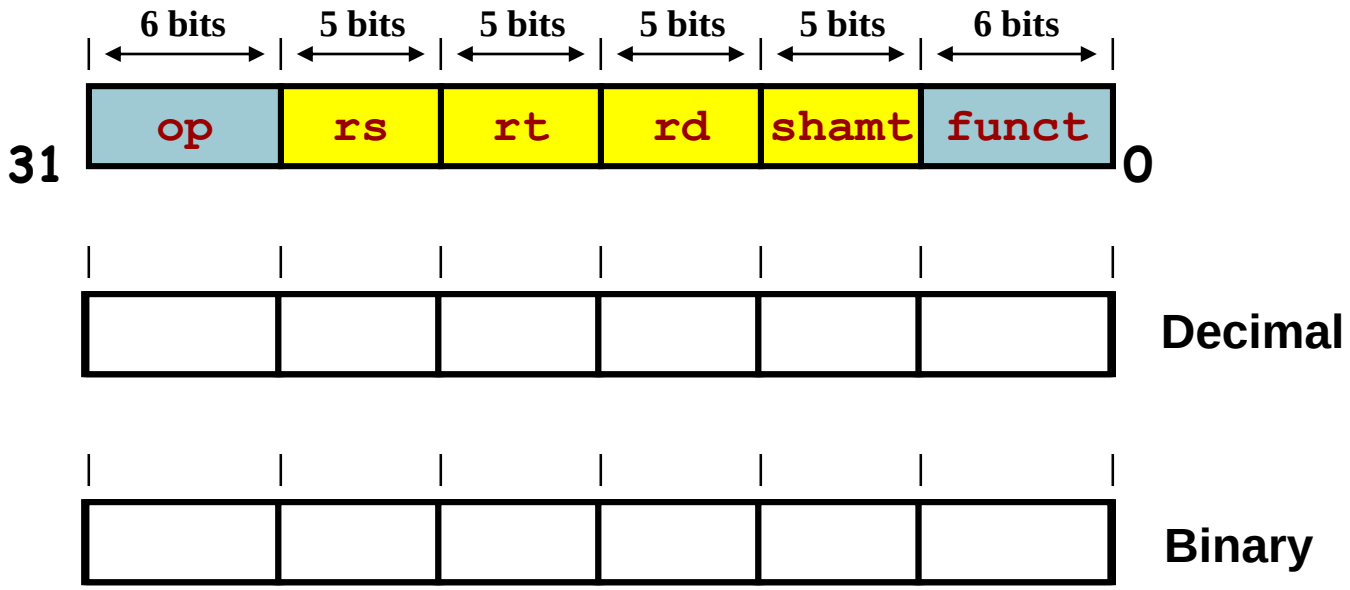
- Used for arithmetic, logical, shift instructions
  - `op`: Basic operation of the instruction (*opcode*)
  - `rs`: first register source operand
  - `rt`: second register source operand
  - `rd`: register destination operand
  - `shamt`: shift amount (more about this later)
  - `funct`: function - specific type of operation
- Also called "R-Format" or "R-Type" Instructions

Arithmetic & Logical Instructions

- Machine language for

  add $8, $17, $18

- See reference card for `op`, `funct` values



MIPS Data Transfer Instructions

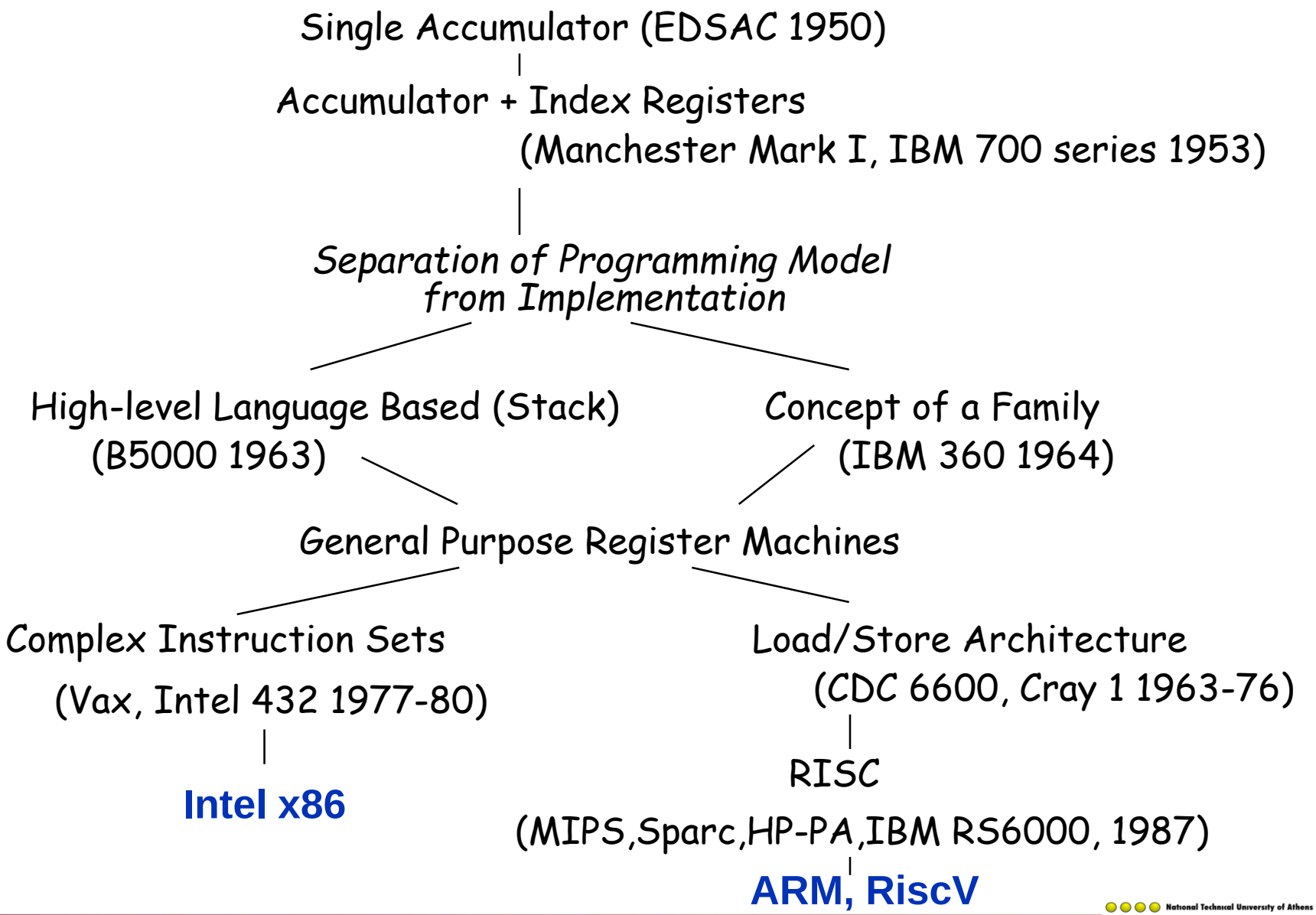
- Transfer data between registers and memory
- Instruction format (assembly)

  lw $dest, offset($addr) load word

  sw $src, offset($addr) store word

- Uses:
  - Accessing a variable in main memory
  - Accessing an array element

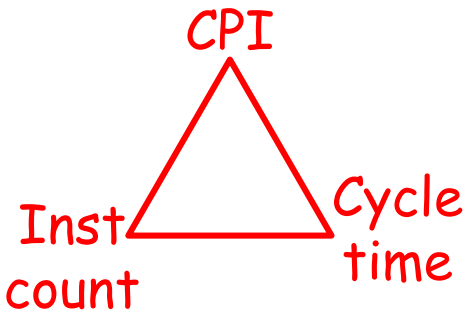
Evolution of Instruction Sets



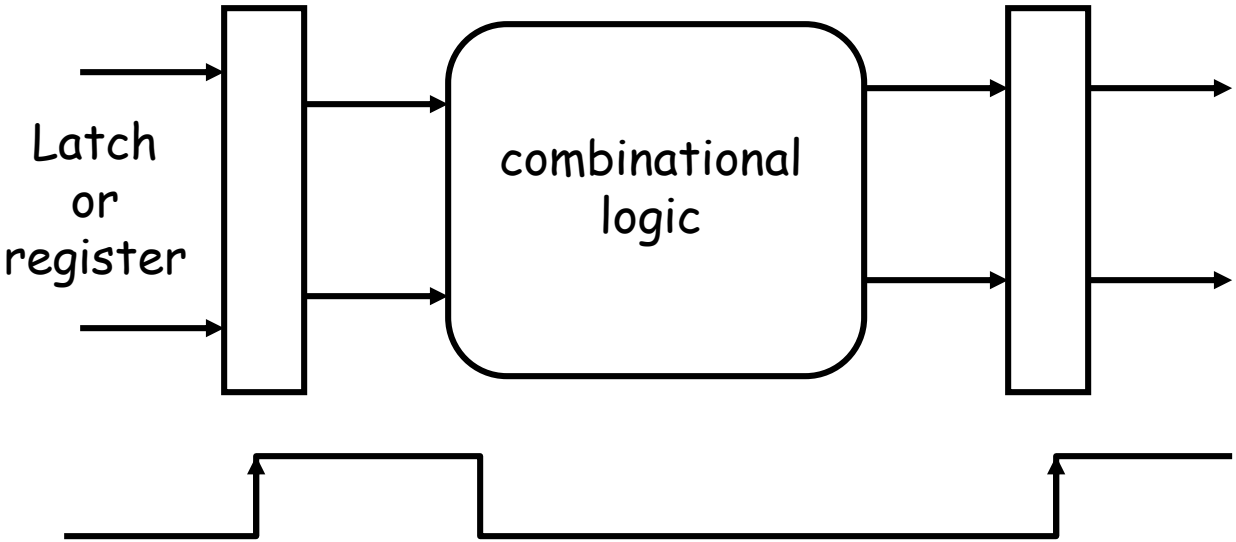
Components of Performance

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

	Instr. count	CPI	Clock rate
Program	X		
Compiler	X	X	
Instruction Set Architecture (ISA)	X	X	X
Οργάνωση		X	X
Τεχνολογία			X



What's a Clock Cycle?



- Old days: 10 FO4 levels of gates (fan-out-of-four)
- Today: determined by numerous time-of-flight issues + gate delays
  - clock propagation, wire lengths, drivers
  - Still ~10-20…

Integrated Approach

What really matters is the functioning of the complete system, I.e. hardware, runtime system, compiler, and operating system

In networking, this is called the "End to End argument"

- Computer architecture is not just about transistors, individual instructions, or particular implementations
- Original RISC projects replaced complex instructions with a compiler + simple instructions

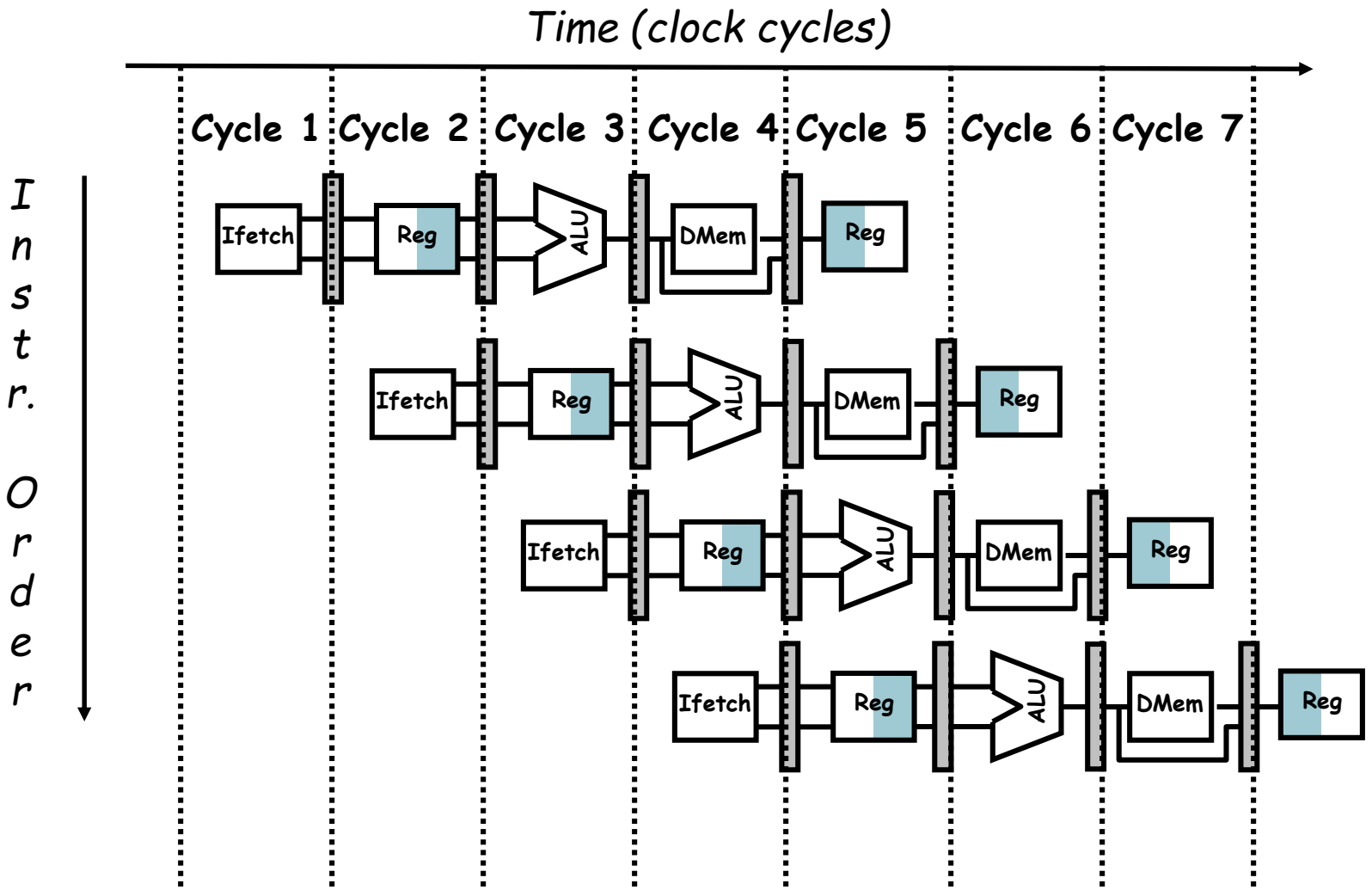
How to achieve better performance?

- ISA is a contract!

- But: underneath the ISA *illusion*….
  - Do more things at once (i.e. in parallel)
  - Do the things that you do faster

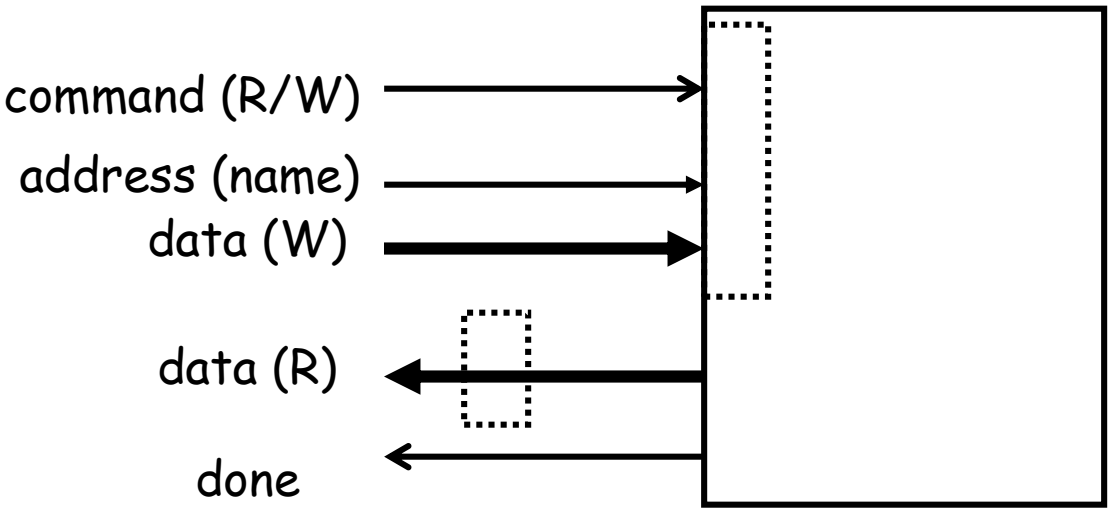
This is what computer architecture is all about!

Example Pipelining

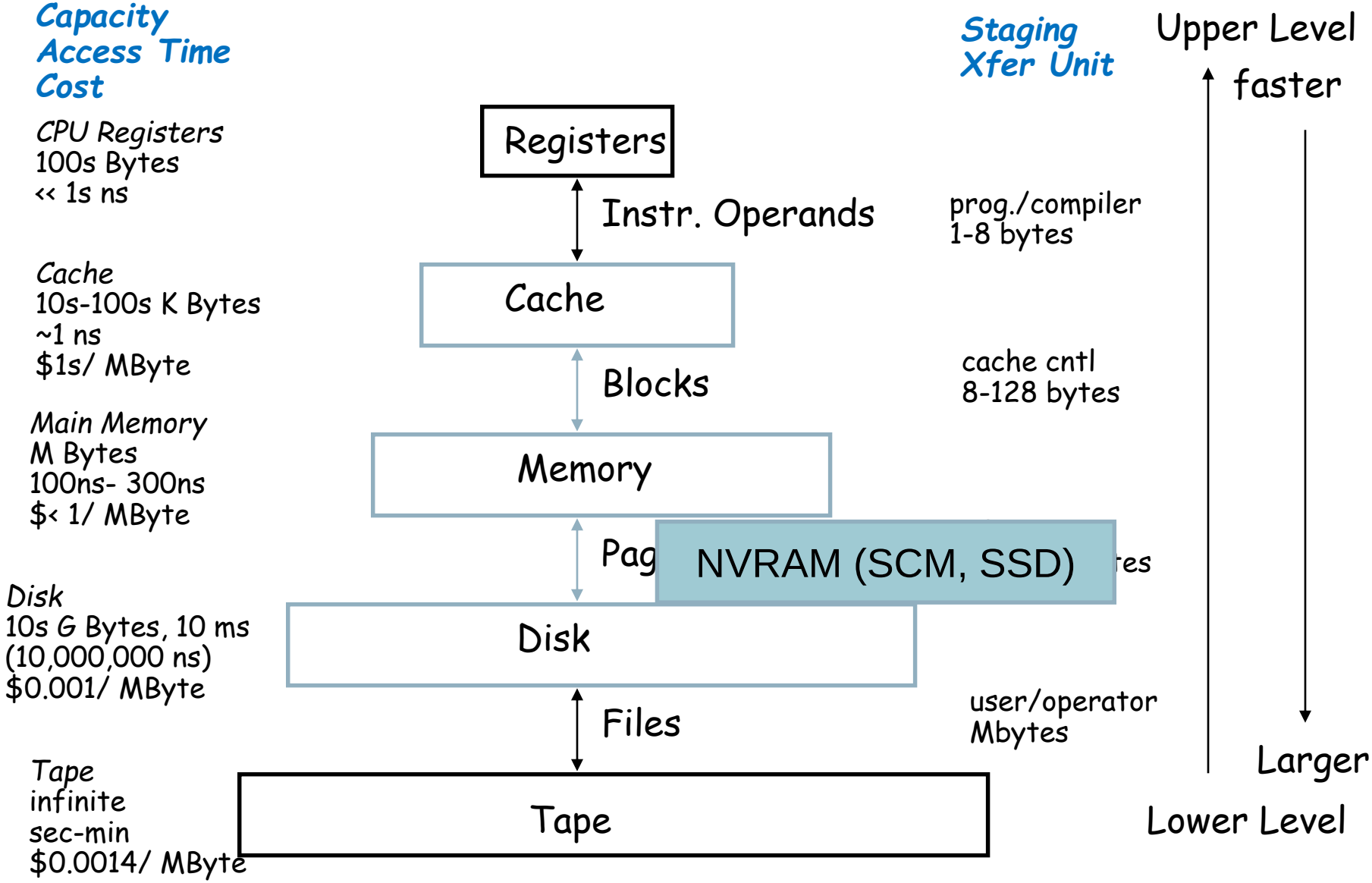


The Memory Abstraction

- Association of <name, value> pairs
  - typically named as byte addresses
  - often values aligned on multiples of size
- Sequence of Reads and Writes
- Write binds a value to an address
- Read of addr returns most recently written value bound to that address



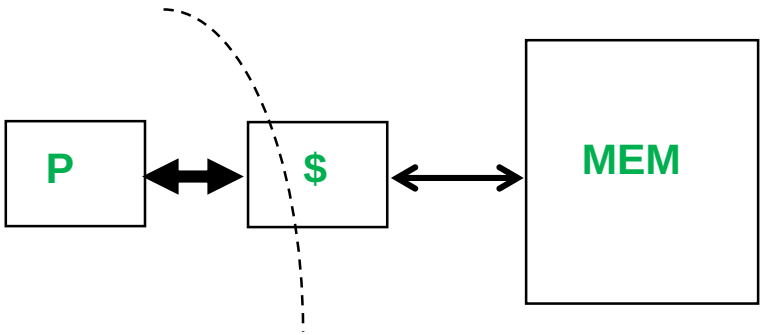
Levels of the Memory Hierarchy



circa 1995 numbers

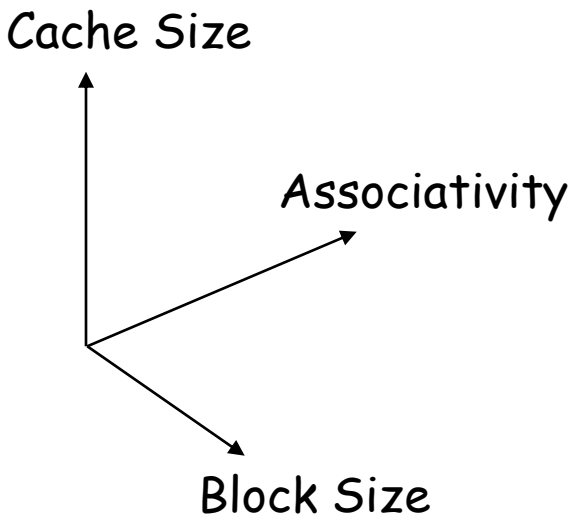
The Principle of Locality

- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time
- Two Different Types of Locality:
  - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
  - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straightline code, array access)
- Last 30 years, HW relied on locality for speed

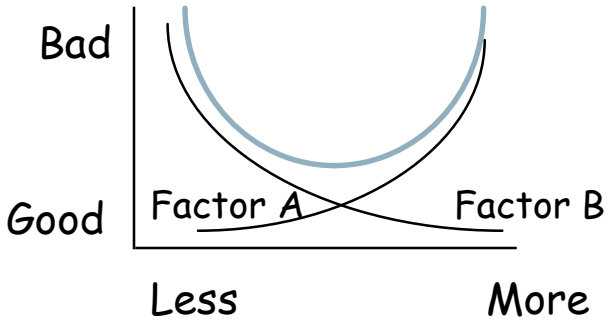


The Cache Design Space

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back



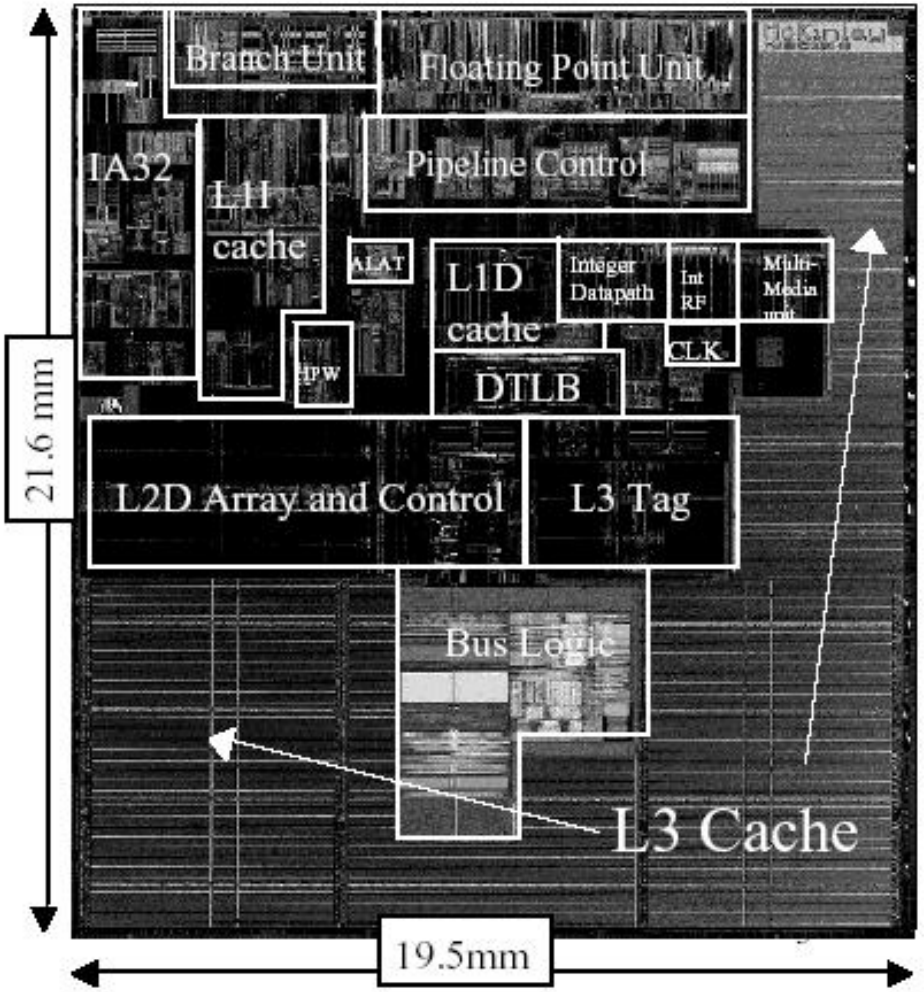
- The optimal choice is a compromise
  - depends on access characteristics
    - workload
    - use (I-cache, D-cache, TLB)
  - depends on technology / cost
- Simplicity often wins



Is it all about memory system design?

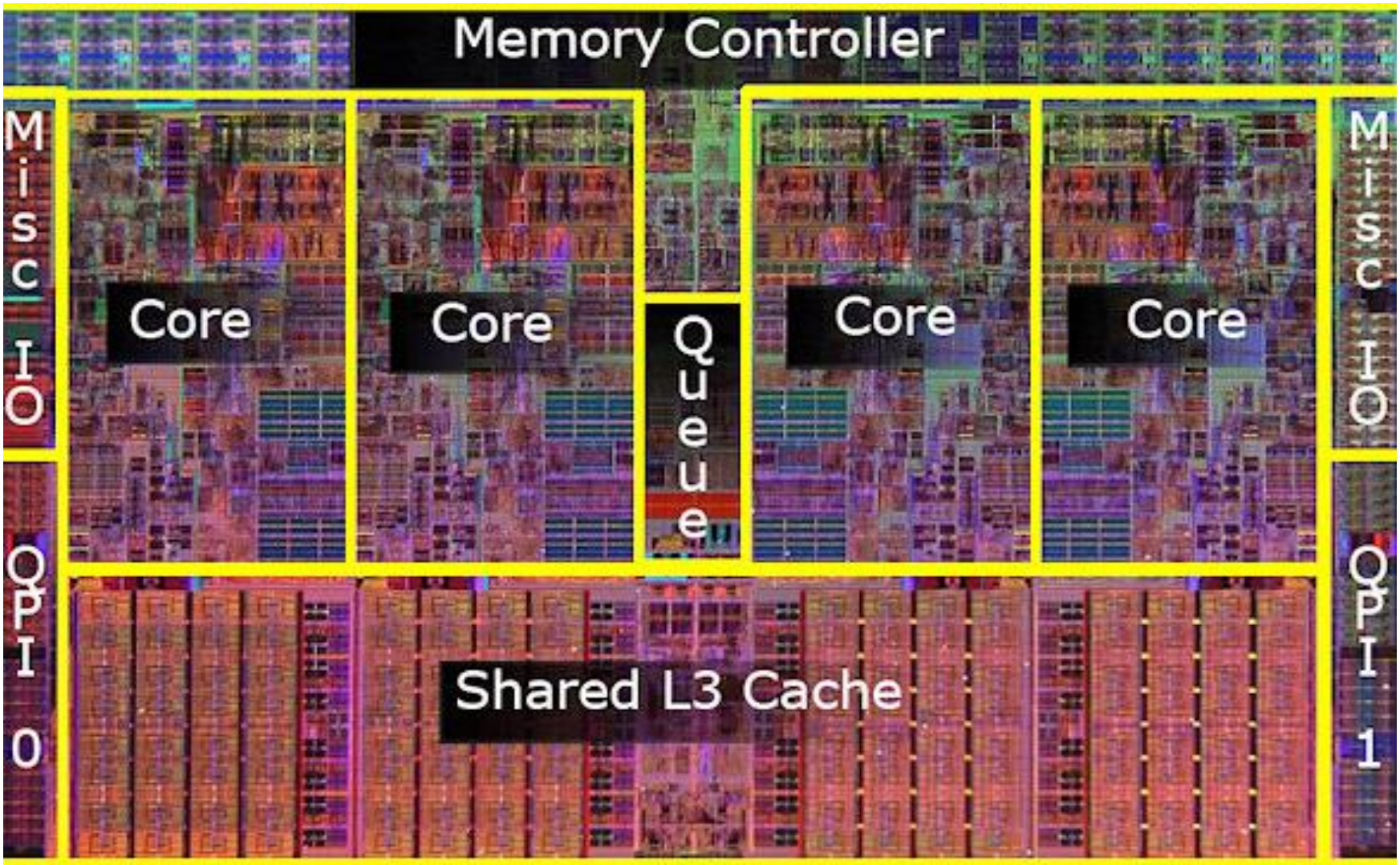
McKinley Floorplan

- 0.18 µm, Al process
- 200MHz system clock
- 1GHz core clock
- Core clocking:
  - 260 mm^2
  - 1 primary driver
  - 5 repeaters
  - 33 delay SLCBs
  - 18k gated buffers
  - 157k clocked latches



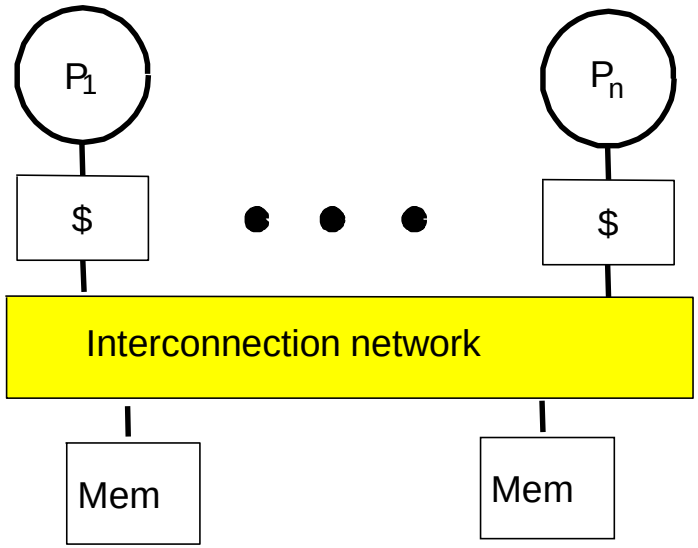
Is it all about memory system design?

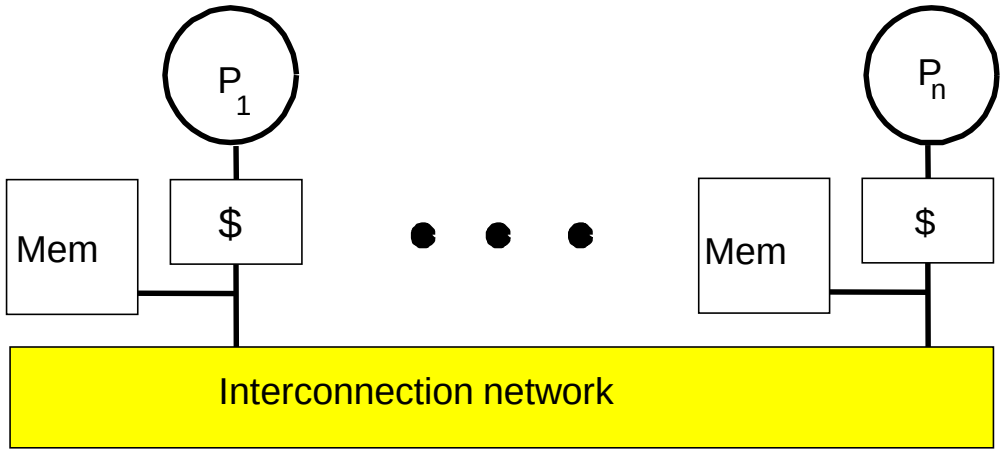
Nahalem, ~2008



Memory Abstraction and Parallelism

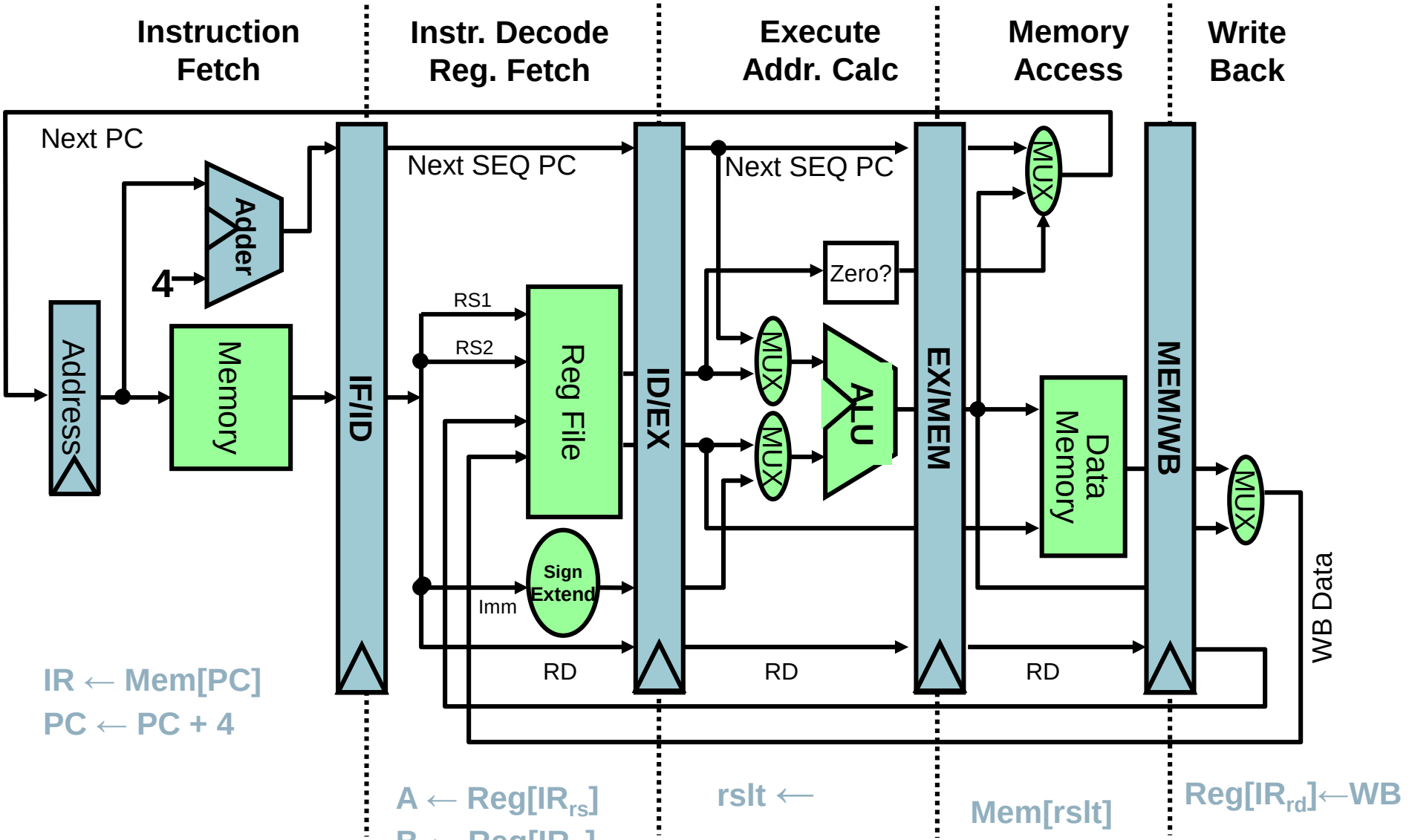
- Maintaining the illusion of sequential access to memory
- What happens when multiple processors access the same memory at once?
  - Do they see a consistent picture?



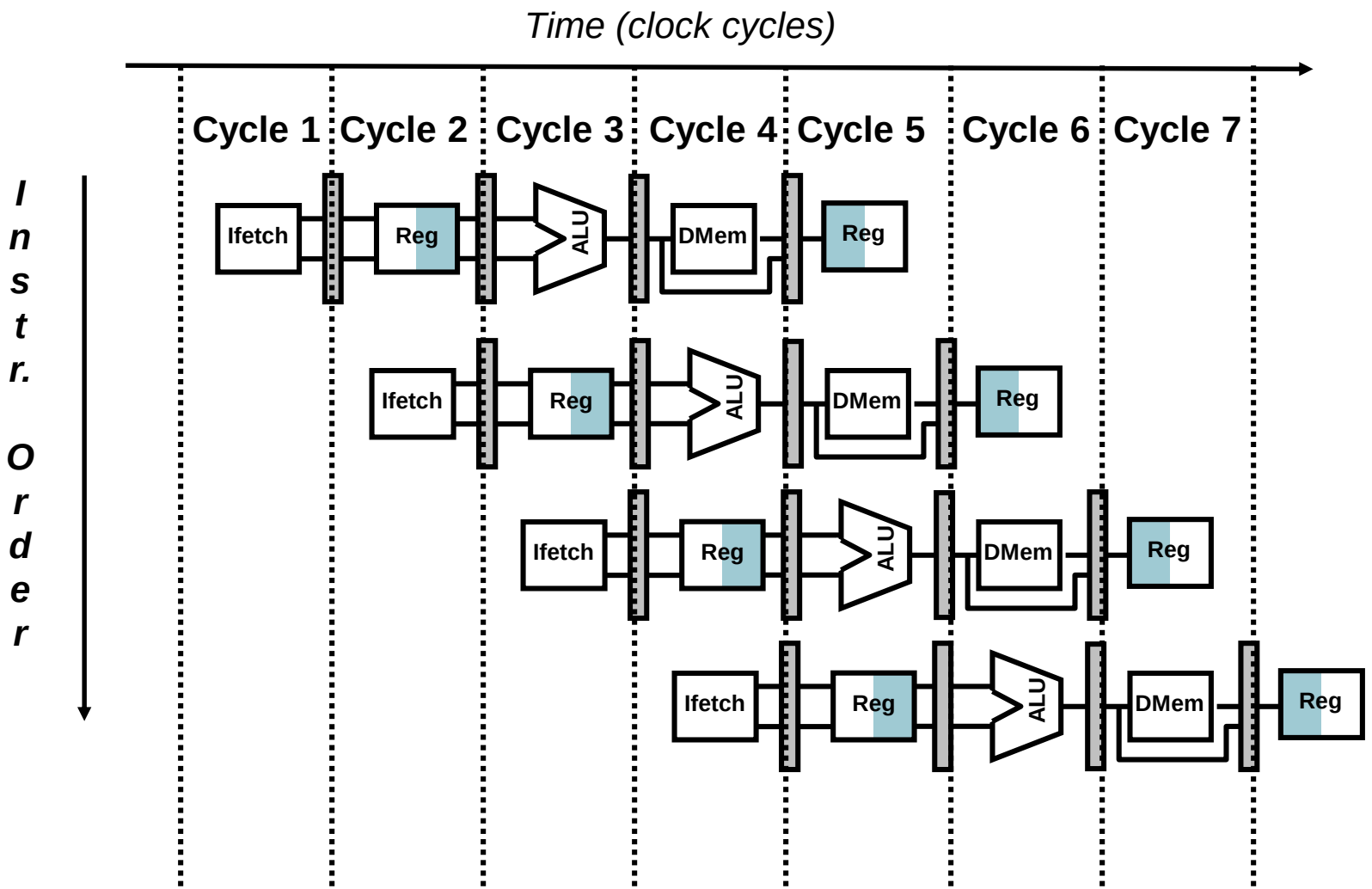


- Processing and processors embedded in the memory?
- Does/should the SW know about hierarchy? NUMA!

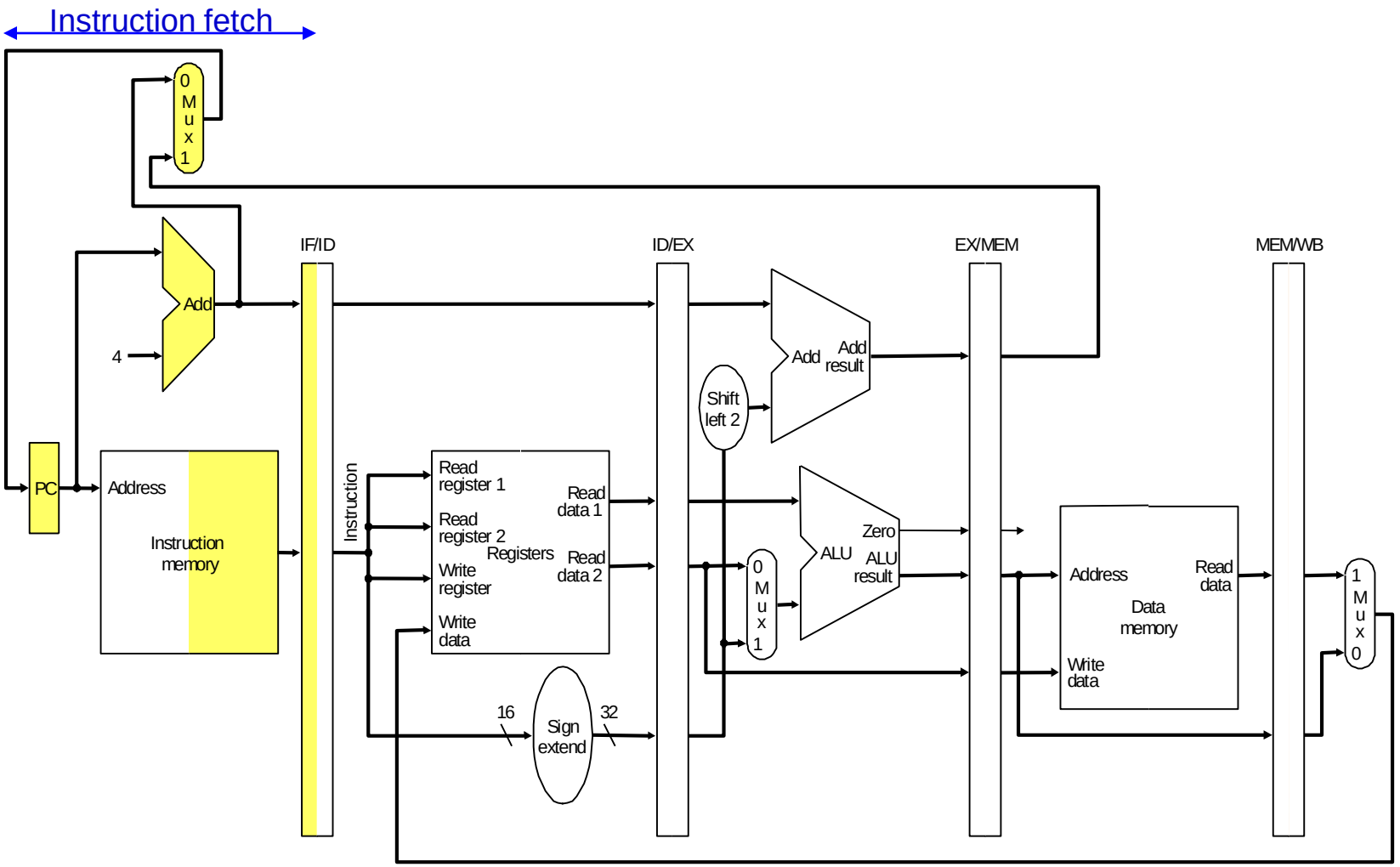
5-Stage Pipelined Datapath



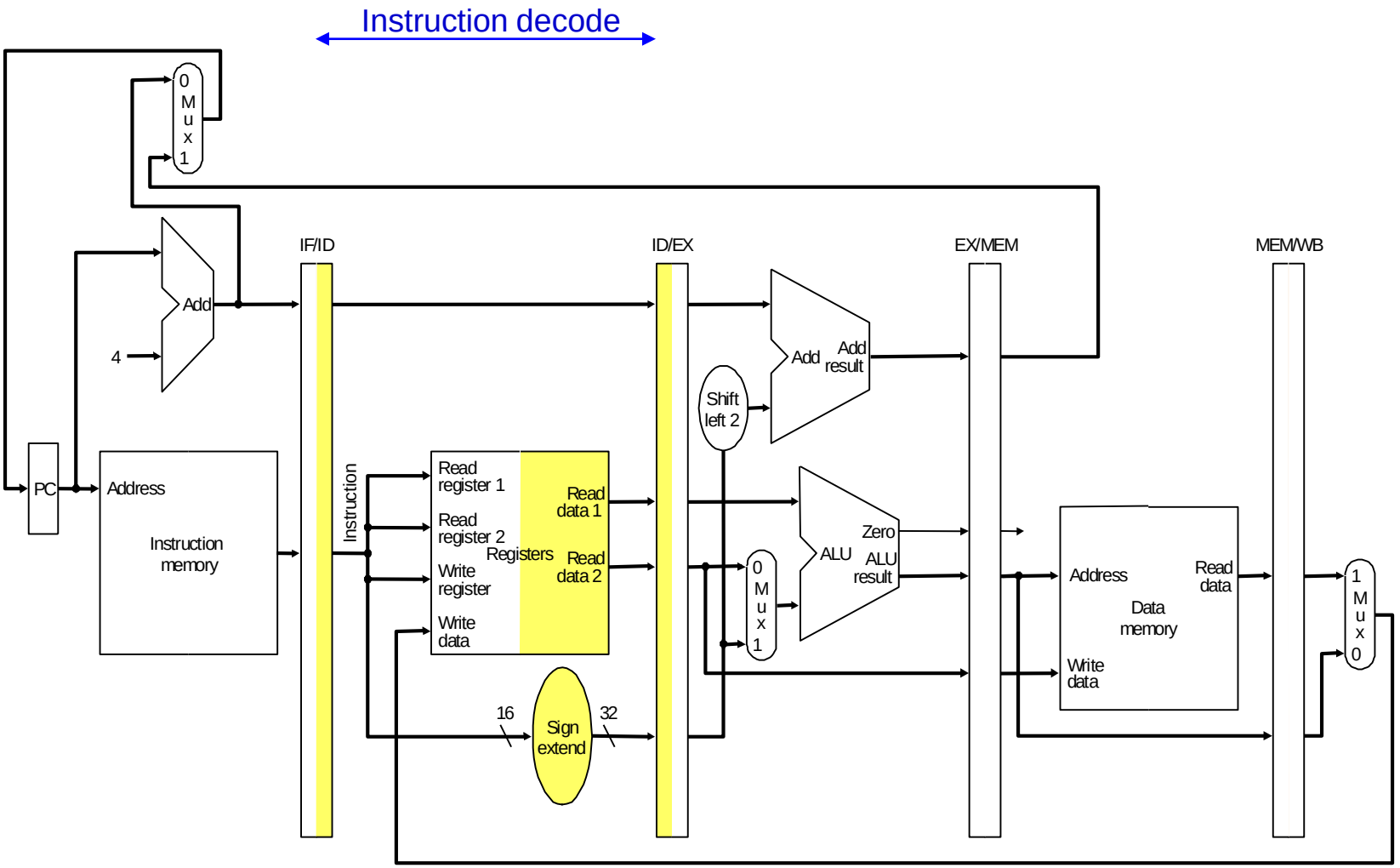
Διάγραμμα χρονισμού



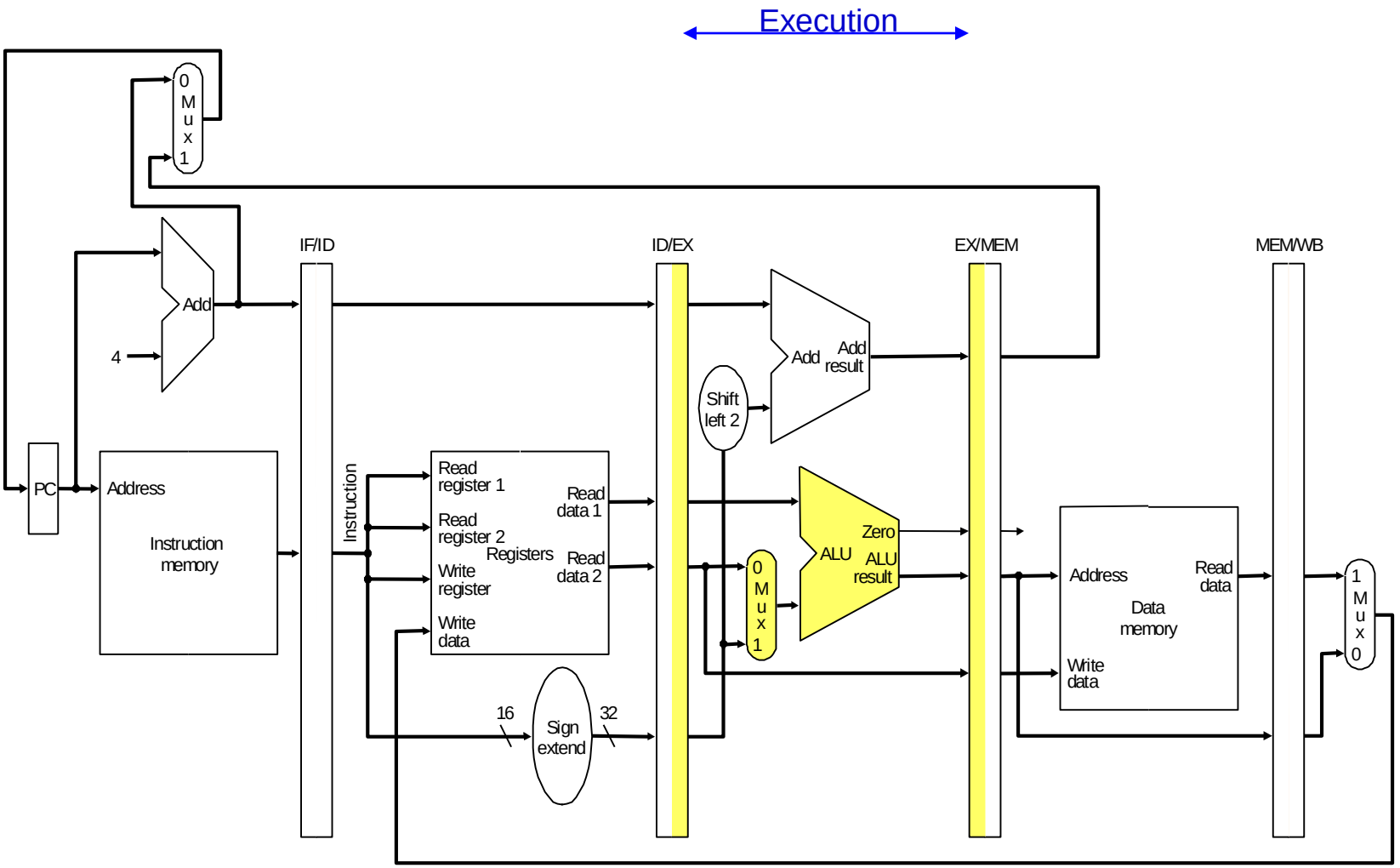
Παράδειγμα για την εντολή lw: Instruction Fetch (IF)



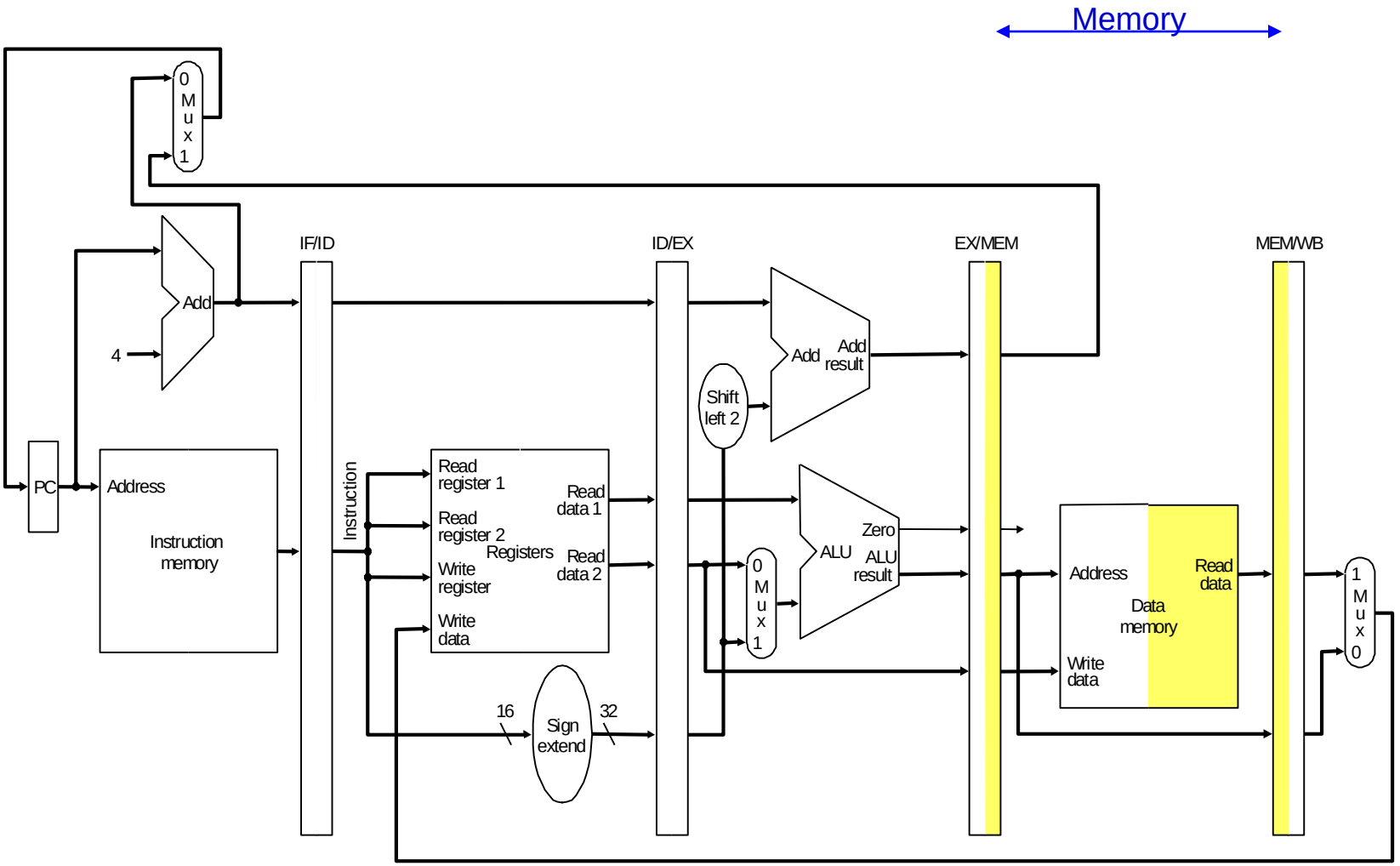
Παράδειγμα για την εντολή lw: Instruction Decode (ID)



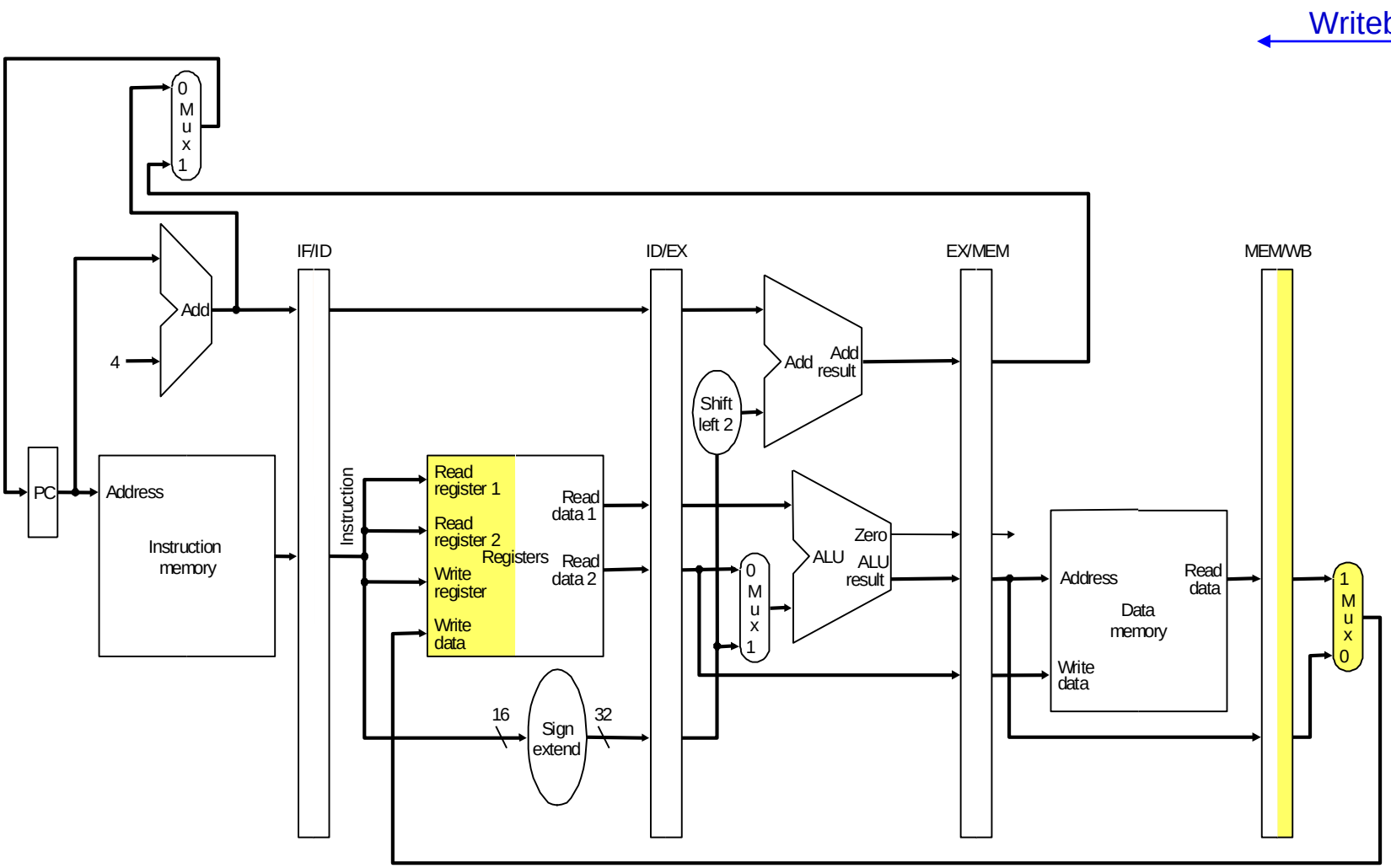
Παράδειγμα για την εντολή lw: Execution (EX)



Παράδειγμα για την εντολή lw: Memory (MEM)



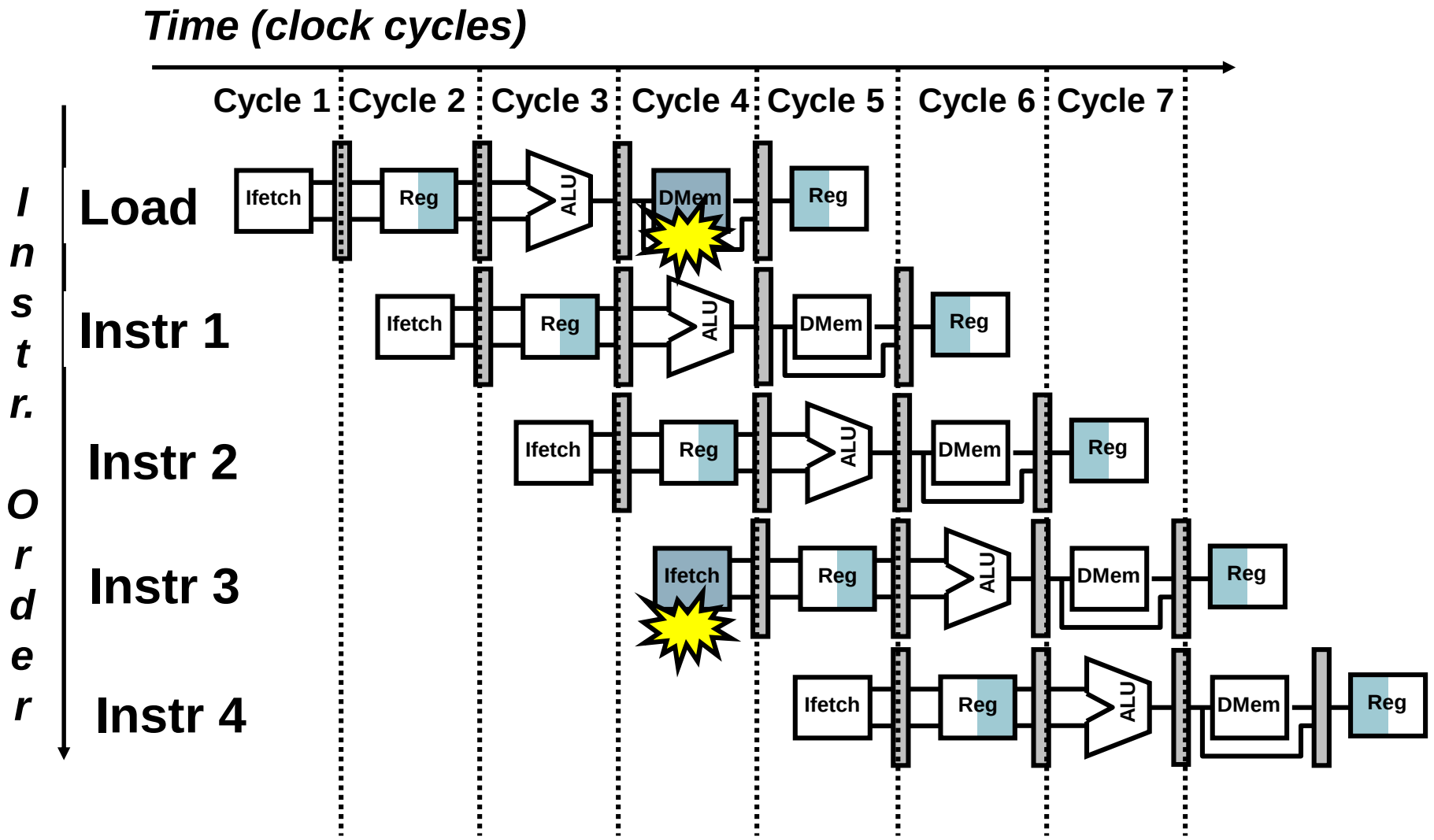
Παράδειγμα για την εντολή lw: Writeback (WB)



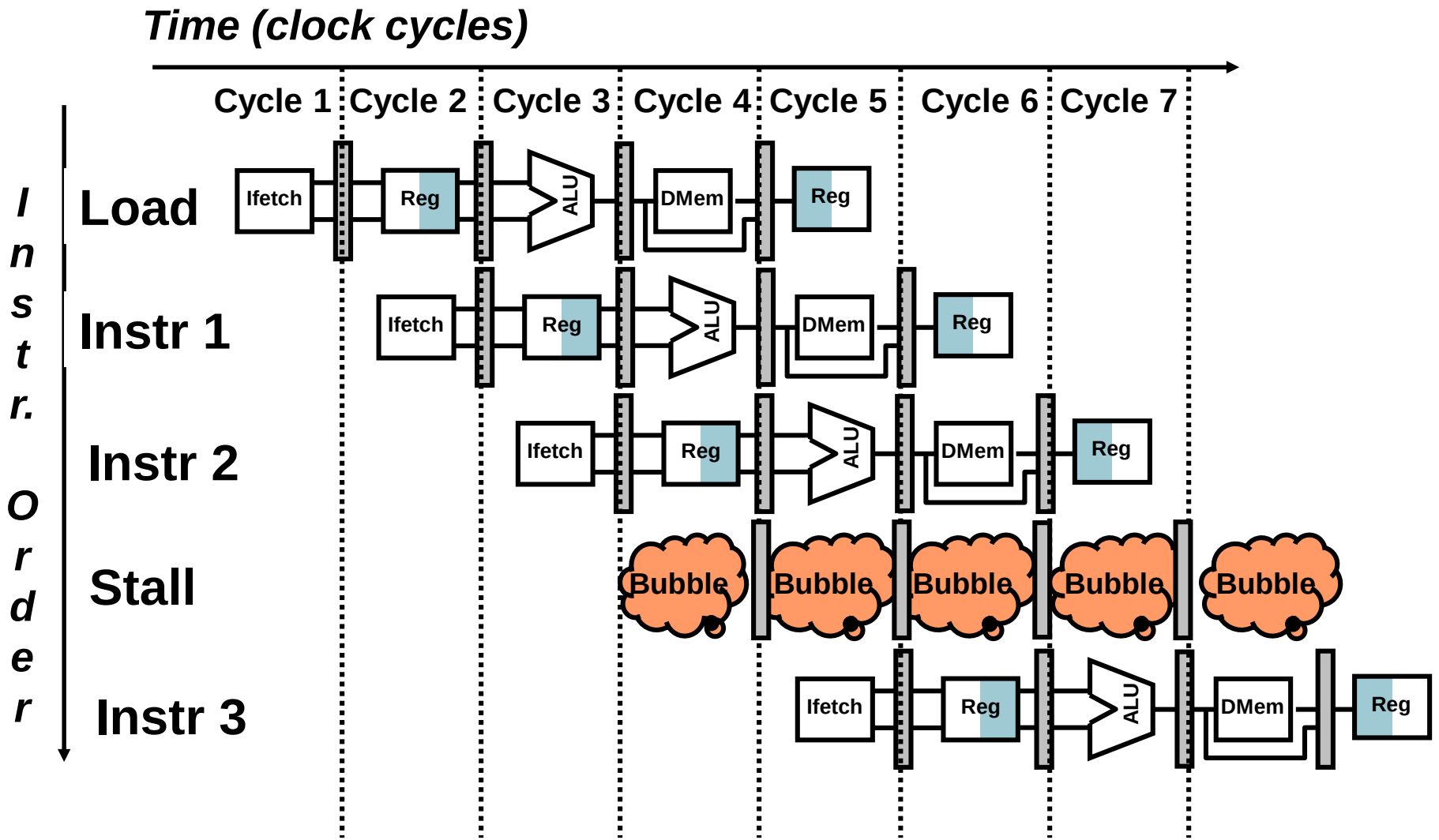
Περιορισμοί pipeline

- Οι κίνδυνοι (hazards) αποτρέπουν την επόμενη εντολή από το να εκτελεστεί στον κύκλο που πρέπει
- Δομικοί κίνδυνοι (structural): όταν το υλικό δεν μπορεί να υποστηρίξει ταυτόχρονη εκτέλεση συγκεκριμένων εντολών
- Κίνδυνοι δεδομένων (data): όταν μια εντολή χρειάζεται το αποτέλεσμα μιας προηγούμενης, η οποία βρίσκεται ακόμη στο pipeline
- Κίνδυνοι ελέγχου (control): όταν εισάγεται καθυστέρηση μεταξύ του φορτώματος εντολών και της λήψης αποφάσεων σχετικά με την αλλαγή της ροής του προγράμματος (branches,jumps)

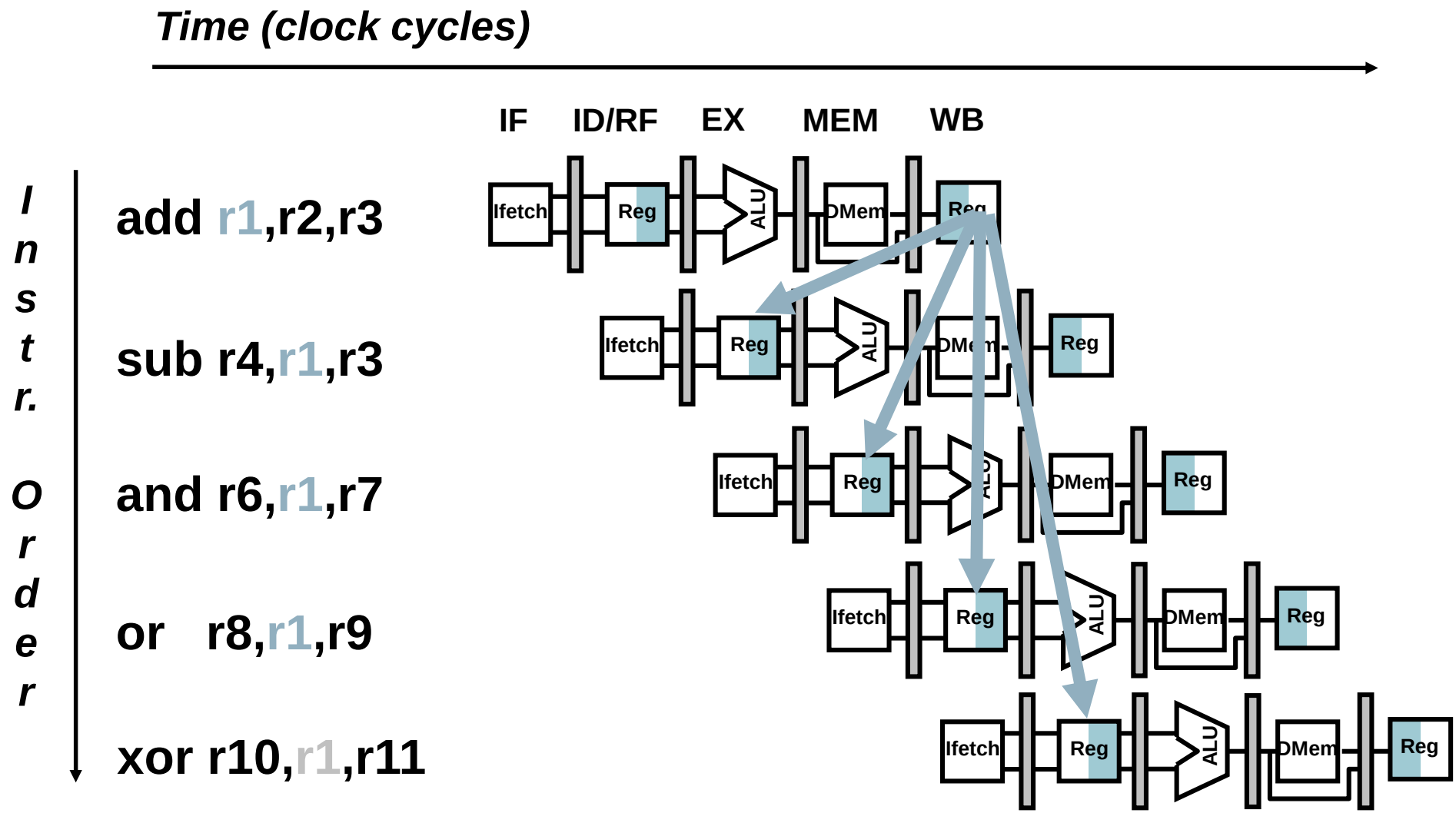
Structural hazard: ένα διαθέσιμο memory port



Structural hazard: επίλυση με stall

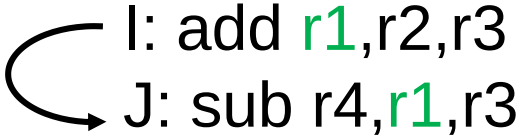


Παράδειγμα data hazard στον r1



Εξαρτήσεις και κίνδυνοι δεδομένων

- Η J είναι data dependent από την I:
  Η J προσπαθεί να διαβάσει τον source operand πριν τον γράψει η I



- ή, η J είναι data dependent από την K, η οποία είναι data dependent από την I (αλυσίδα εξαρτήσεων)
- Πραγματικές εξαρτήσεις (True Dependences)
- Προκαλούν Read After Write (RAW) hazards στο pipeline

Εξαρτήσεις και κίνδυνοι δεδομένων

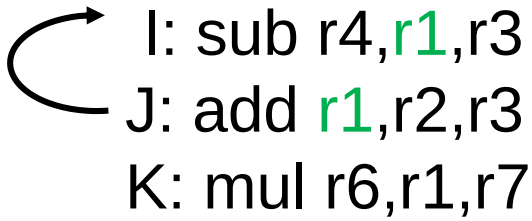
- Οι εξαρτήσεις είναι ιδιότητα των προγραμμάτων
- Η παρουσία μιας εξάρτησης υποδηλώνει την πιθανότητα εμφάνισης hazard, αλλά το αν θα συμβεί πραγματικά το hazard, και το πόση καθυστέρηση θα εισάγει, είναι ιδιότητα του pipeline
- Η σημασία των εξαρτήσεων δεδομένων:

1) υποδηλώνουν την πιθανότητα για hazards
2) καθορίζουν τη σειρά σύμφωνα με την οποία πρέπει να υπολογιστούν τα δεδομένα
3) θέτουν ένα άνω όριο στο ποσό του παραλληλισμού που μπορούμε να εκμεταλλευτούμε

Name Dependences (1): Anti-dependences

Name dependence, όταν 2 εντολές χρησιμοποιούν τον ίδιο καταχωρητή ή θέση μνήμης (”name”), χωρίς όμως να υπάρχει πραγματική ροή δεδομένων μεταξύ τους

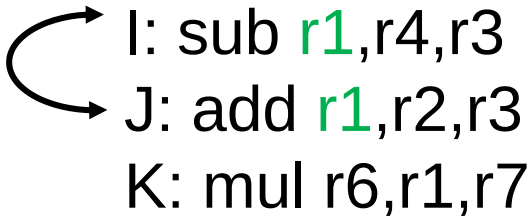
1. Anti-dependence: η J γράφει τον r1 πριν τον διαβάσει η I



- Προκαλεί Write After Read (WAR) data hazards στο pipeline
- Δε μπορεί να συμβεί στο κλασικό 5-stage pipeline διότι:
  - όλες οι εντολές χρειάζονται 5 κύκλους για να εκτελεστούν, και
  - οι αναγνώσεις συμβαίνουν πάντα στο στάδιο 2, και
  - οι εγγραφές στο στάδιο 5

Name Dependences (2):Output dependences

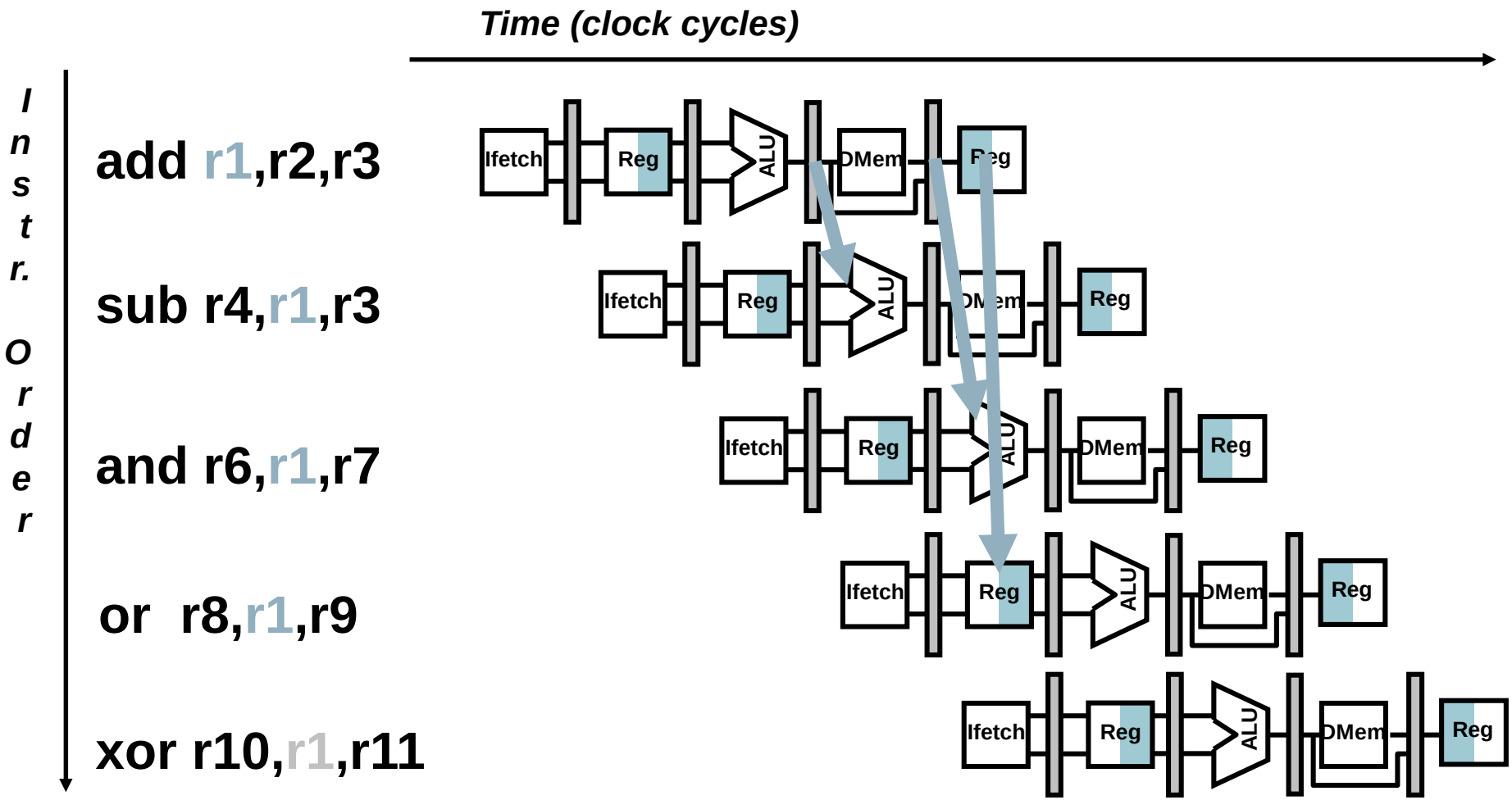
2. Output dependence: η J γράφει τον r1 πριν τον γράψει η I



- Προκαλεί Write After Write (WAW) data hazards στο pipeline
- Δε μπορεί να συμβεί στο κλασικό 5-stage pipeline διότι:
  - όλες οι εντολές χρειάζονται 5 κύκλους για να εκτελεστούν, και
  - οι εγγραφές συμβαίνουν πάντα στο στάδιο 5

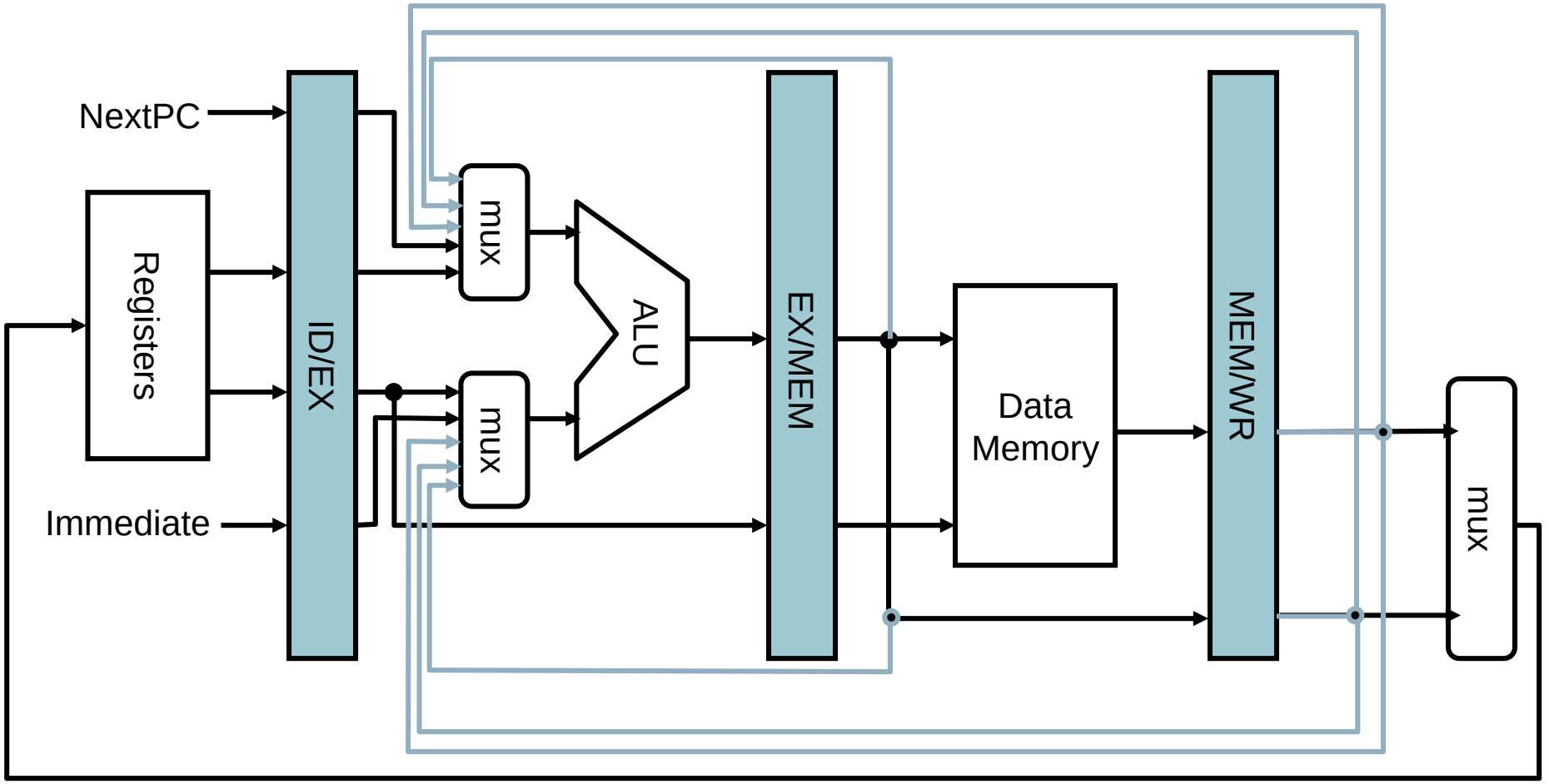
- Τόσο οι WAW όσο και οι WAR κίνδυνοι συναντώνται σε πιο περίπλοκα pipelines (π.χ. multi-cycle pipeline, out-of-order execution)

Προώθηση



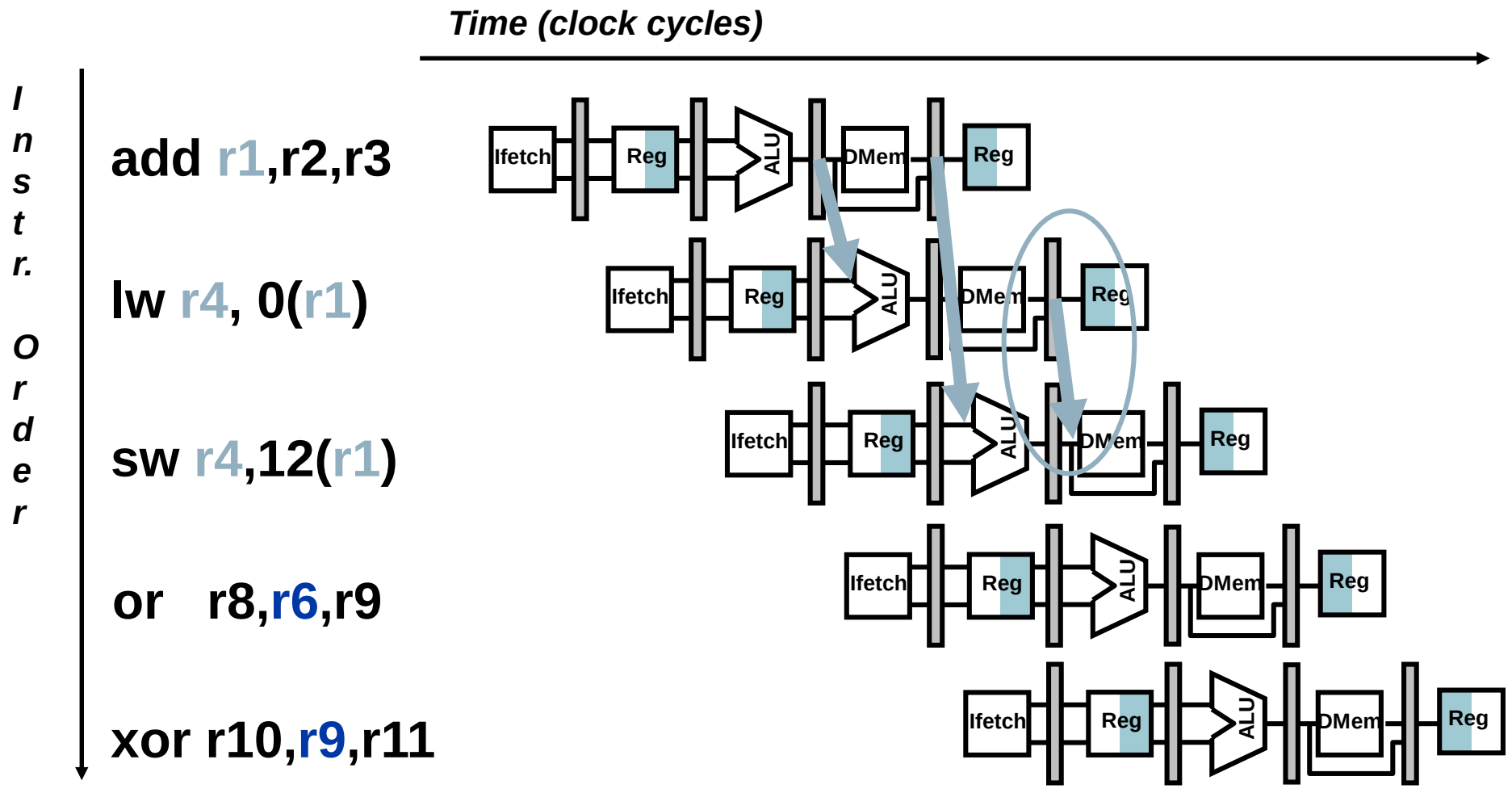
- μειώνονται τα RAW hazards

Αλλαγές στο hardware για την υποστήριξη προώθησης

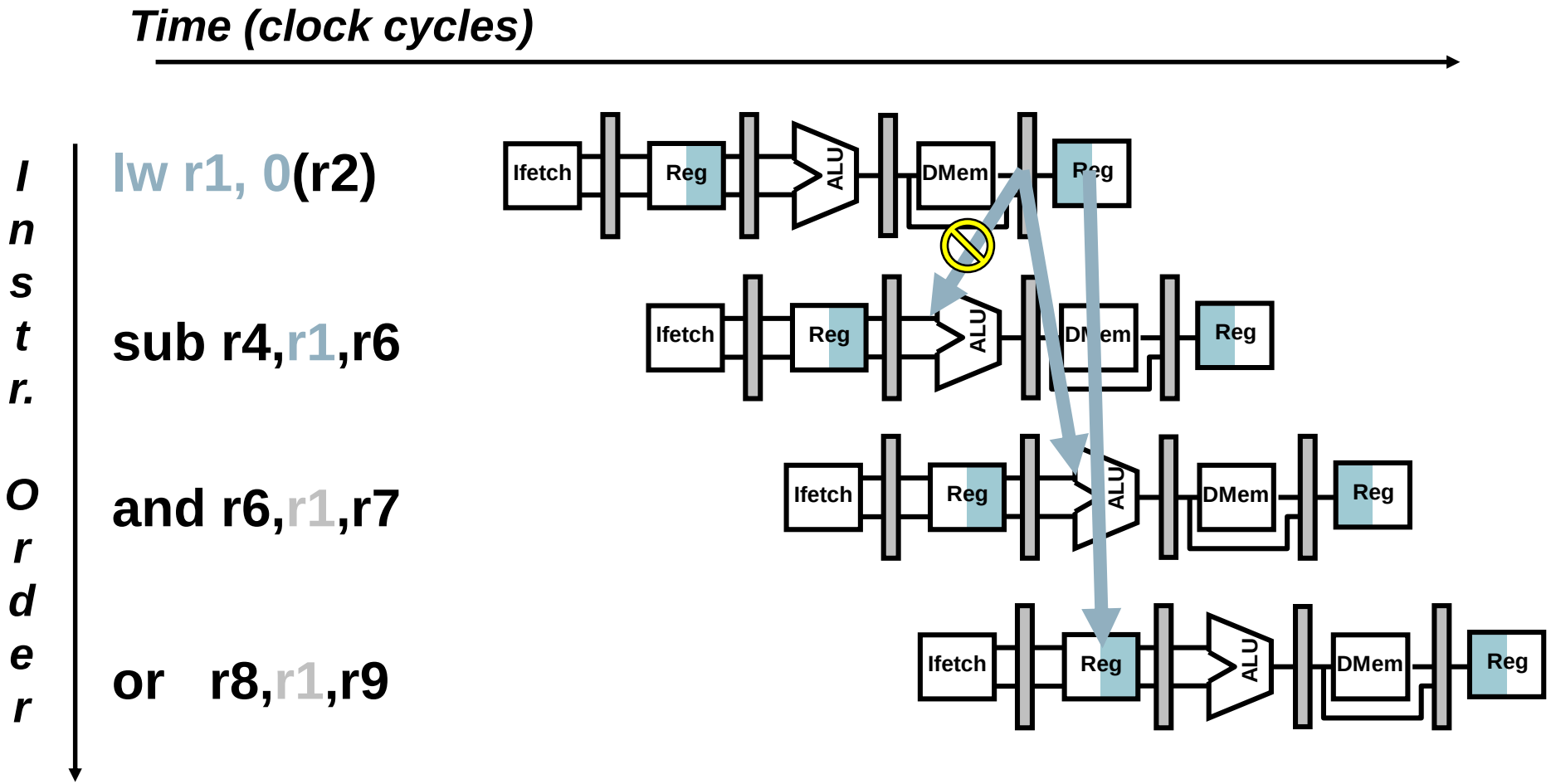


Προώθηση EX->EX, MEM->EX

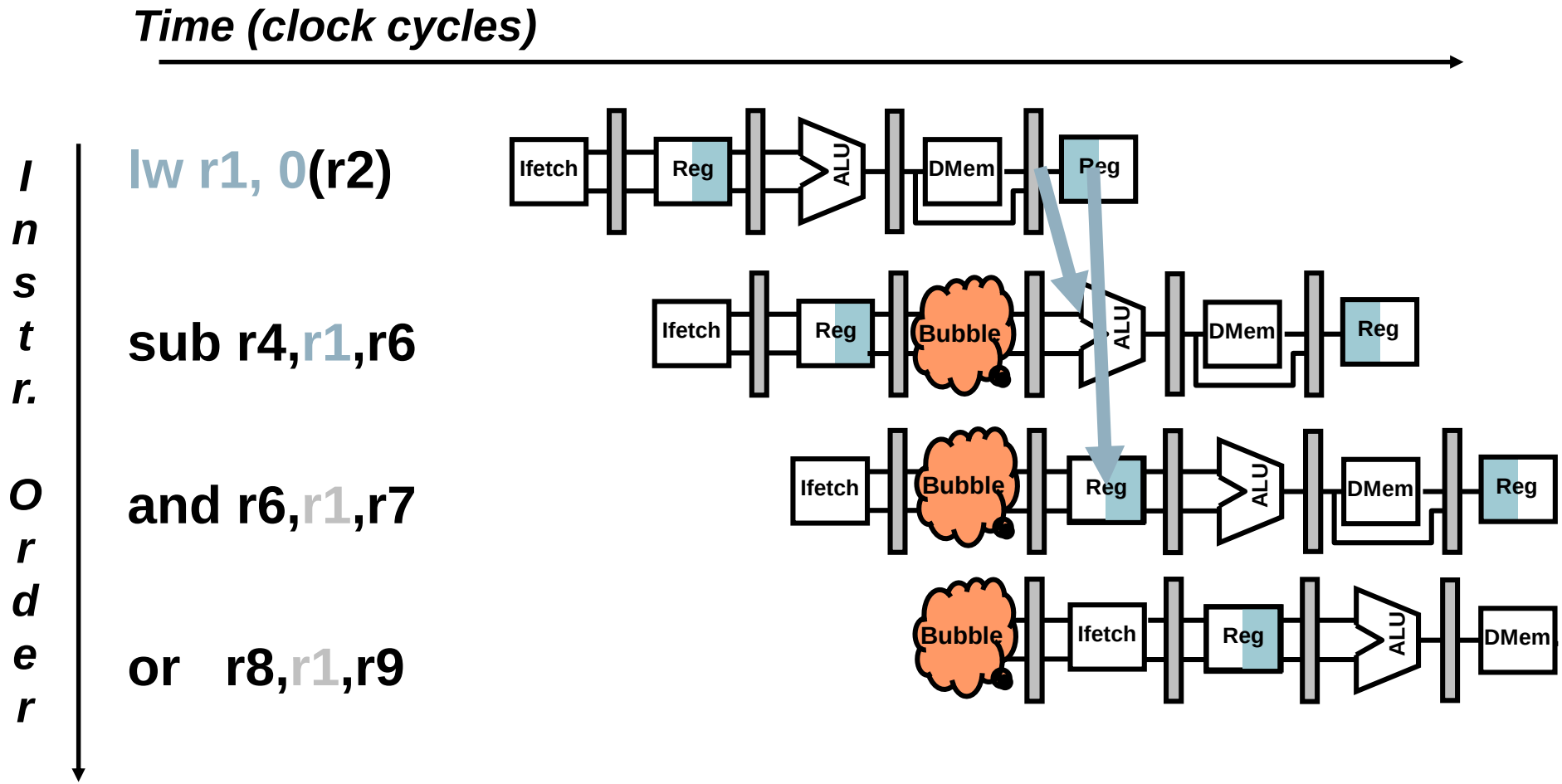
Προώθηση MEM->MEM



Η προώθηση δεν δουλεύει πάντα!



Η προώθηση δεν δουλεύει πάντα!



Αναδιάταξη εντολών για αποφυγή RAW hazards

Πώς μπορούμε να παράγουμε γρηγορότερο κώδικα assembly για τις ακόλουθες πράξεις?

a = b + c;

d = e – f;

Slow code:

```
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
```

Fast code:

```
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
```

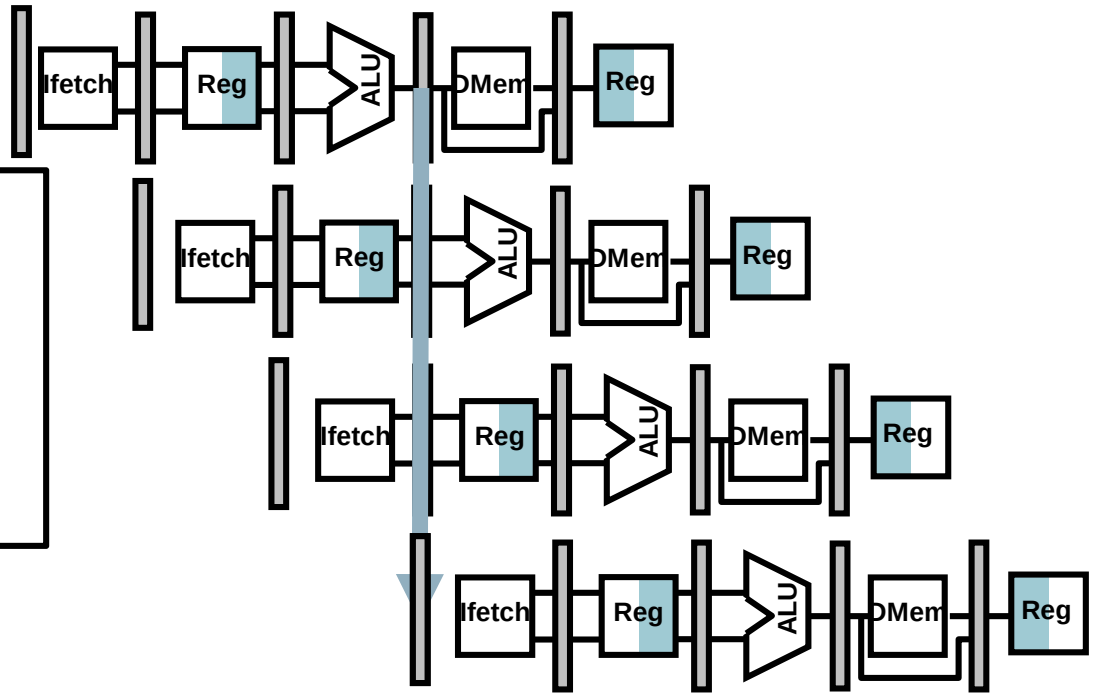
Κίνδυνοι ελέγχου στις εντολές διακλάδωσης: stalls 2 σταδίων

10: beq r1,r3,36

14: and r2,r3,r5

18: or r6,r1,r7

36: xor r10,r1,r11

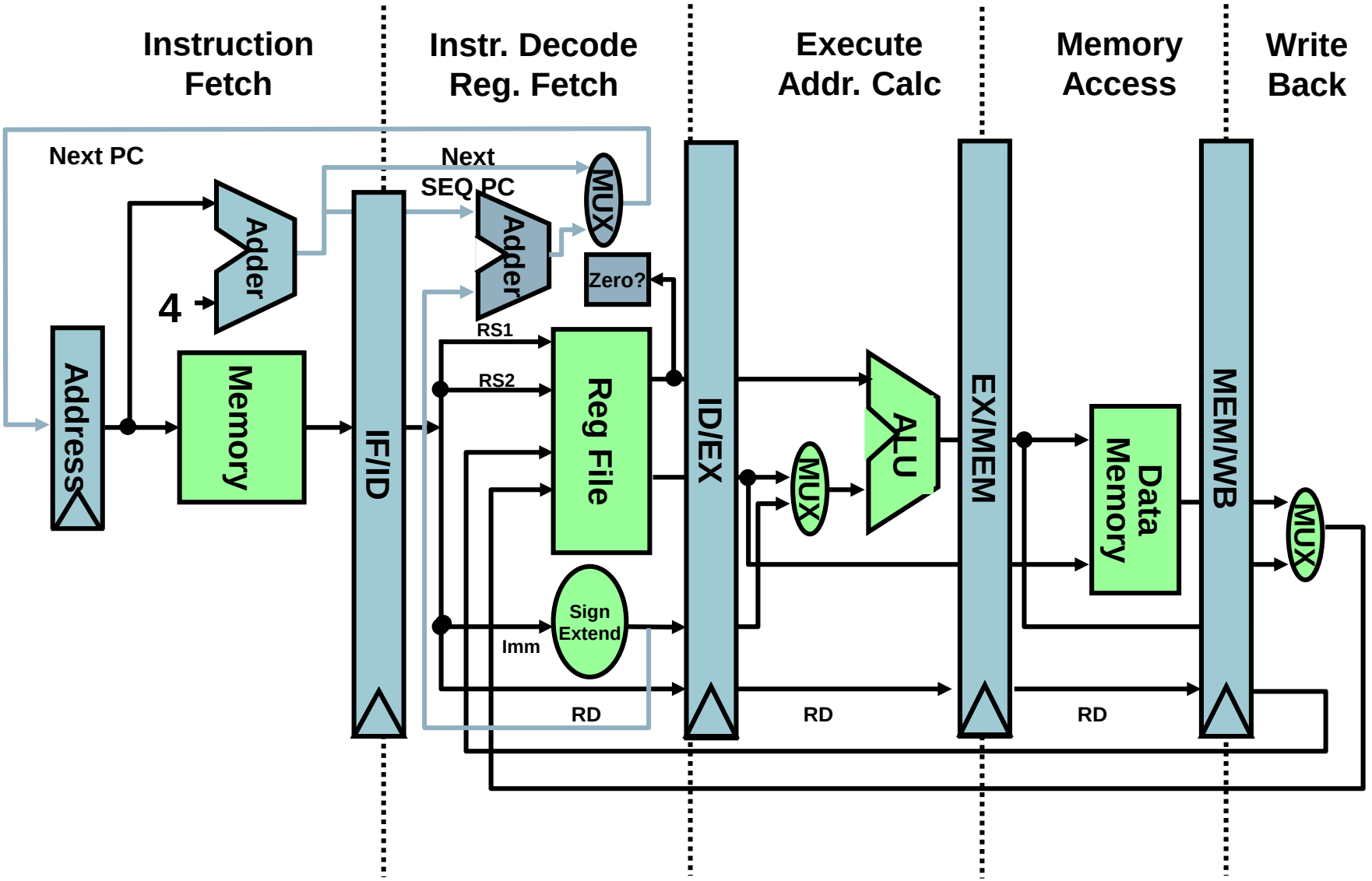


Επίπτωση των branch stalls

- Αν CPI = 1, η συχνότητα των branches 30%, και τα branch stalls διαρκούν 2 κύκλους => νέο CPI = 1.6!

- Μια λύση: καθόρισε το αποτέλεσμα του branch νωρίτερα, ΚΑΙ υπολόγισε τη διεύθυνση-στόχο του branch νωρίτερα
  - μετακίνηση του ελέγχου ισότητας ενός καταχωρητή με το 0 στο στάδιο ID/RF
  - προσθήκη αθροιστή στο στάδιο ID/RF για τον υπολογισμό του PC της διεύθυνσης-στόχου
  - 1 κύκλος branch penalty έναντι 2

Τροποποιήσεις στο pipeline



Αντιμετώπιση control hazards: 4 εναλλακτικές

#1: Πάγωμα του pipeline μέχρι ο στόχος του branch να γίνει γνωστός

#2: Πρόβλεψη “Not Taken” για κάθε branch

- συνεχίζουμε να φορτώνουμε τις επόμενες εντολές, σα να ήταν η εντολή branch μια «κανονική» εντολή
- απορρίπτουμε από το pipeline αυτές τις εντολές, αν τελικά το branch είναι “Taken”
- το PC+4 είναι ήδη υπολογισμένο, το χρησιμοποιούμε για να πάρουμε την επόμενη εντολή

#3: Πρόβλεψη “Taken” για κάθε branch

- φορτώνουμε εντολές αρχίζοντας από τη διεύθυνση-στόχο του branch
- Στον MIPS η διεύθυνση-στόχος δε γίνεται γνωστή πριν το αποτέλεσμα του branch
  - κανένα επιπλέον πλεονέκτημα στον MIPS (1 cycle branch penalty)
  - θα είχε νόημα σε άλλα pipelines, όπου η διεύθυνση-στόχος γίνεται γνωστή πριν το αποτέλεσμα του branch

Αντιμετώπιση control hazards: 4 εναλλακτικές

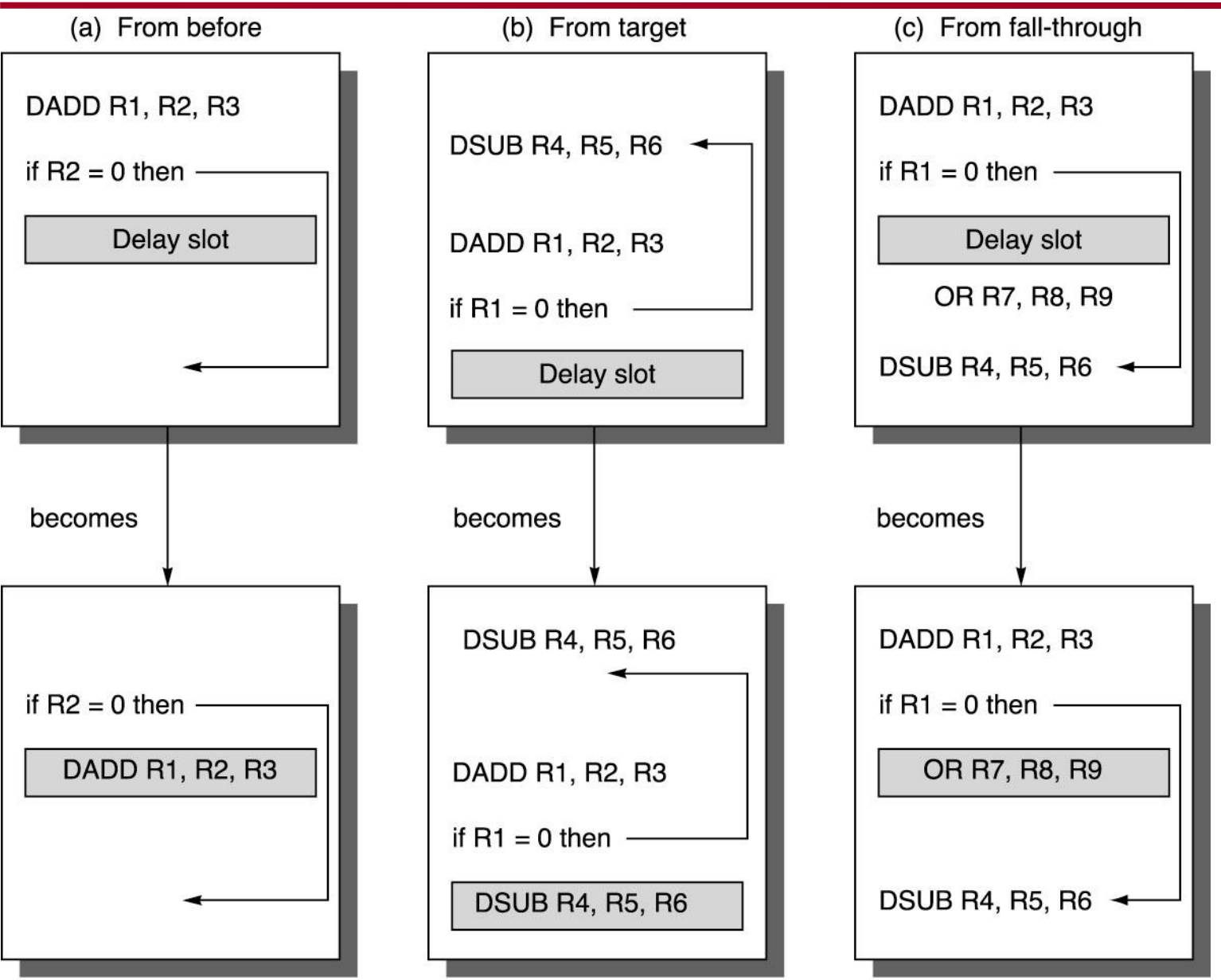
#4: Delayed Branches

```
branch instruction
  sequential successor1
  sequential successor2
  ........
  sequential successorn
branch target if taken
```

Branch delay μήκους n:
οι εντολές εκτελούνται είτε το branch είναι Taken είτε όχι

- delay ενός slot: επιτρέπει απόφαση και υπολογισμό διεύθυνσης-στόχου στο 5-stage pipeline χωρίς stalls

«Δρομολόγηση» ενός branch delay slot



καλύτερη επιλογή;

(α) ✓

(β) ~

(γ) οκ

Γιατί;

Περιορισμοί των βαθμωτών αρχιτεκτονικών

- Μέγιστο throughput: 1 εντολή/κύκλο ρολογιού (IPC≤1)
- Υποχρεωτική ροή όλων των (διαφορετικών) τύπων εντολών μέσα από κοινή σωλήνωση
- Εισαγωγή καθυστερήσεων σε ολόκληρη την ακολουθία εκτέλεσης λόγω stalls μίας εντολής (οι απόλυτα βαθμωτές αρχιτεκτονικές πραγματοποιούν εν σειρά (in-order) εκτέλεση των εντολών)

Πώς μπορούν να ξεπεραστούν οι περιορισμοί;

- Εκτέλεση πολλαπλών εντολών ανά κύκλο μηχανής (παράλληλη εκτέλεση)

  → υπερβαθμωτές αρχιτεκτονικές
- Ενσωμάτωση διαφορετικών αγωγών ροής δεδομένων, ο καθένας με όμοιες (πολλαπλή εμφάνιση του ίδιου τύπου) ή και ετερογενείς λειτουργικές μονάδες

  → multicycle operations
- Δυνατότητα εκτέλεσης εκτός σειράς (out-of-order) των εντολών

  → δυναμικές αρχιτεκτονικές

Εναλλακτικά...

Pipeline CPI =

Ideal pipeline CPI +

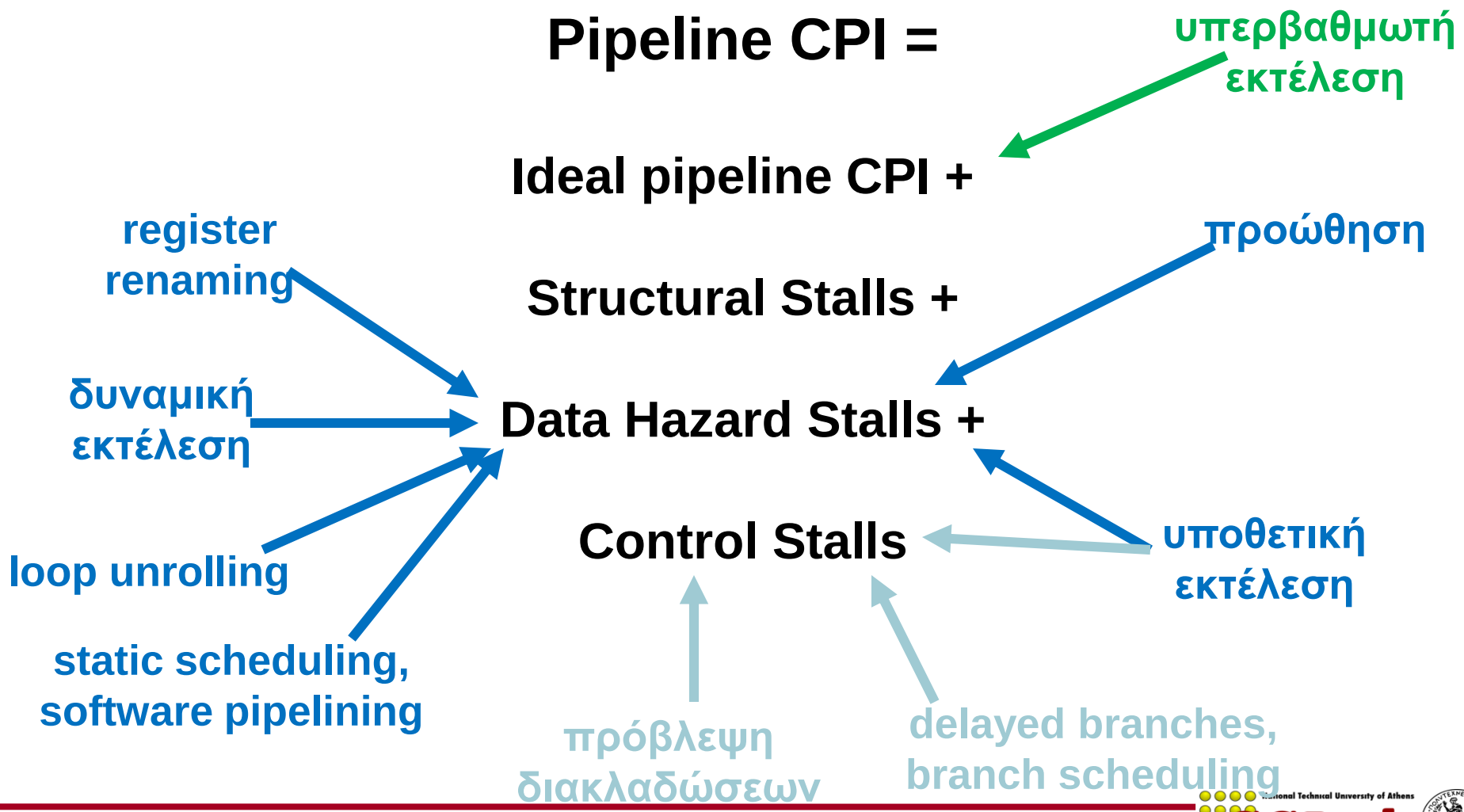
Structural Stalls +

Data Hazard Stalls +

Control Stalls

μέτρο της μέγιστης απόδοσης που μπορούμε να έχουμε με την εκάστοτε υλοποίηση του pipeline

Εναλλακτικά...

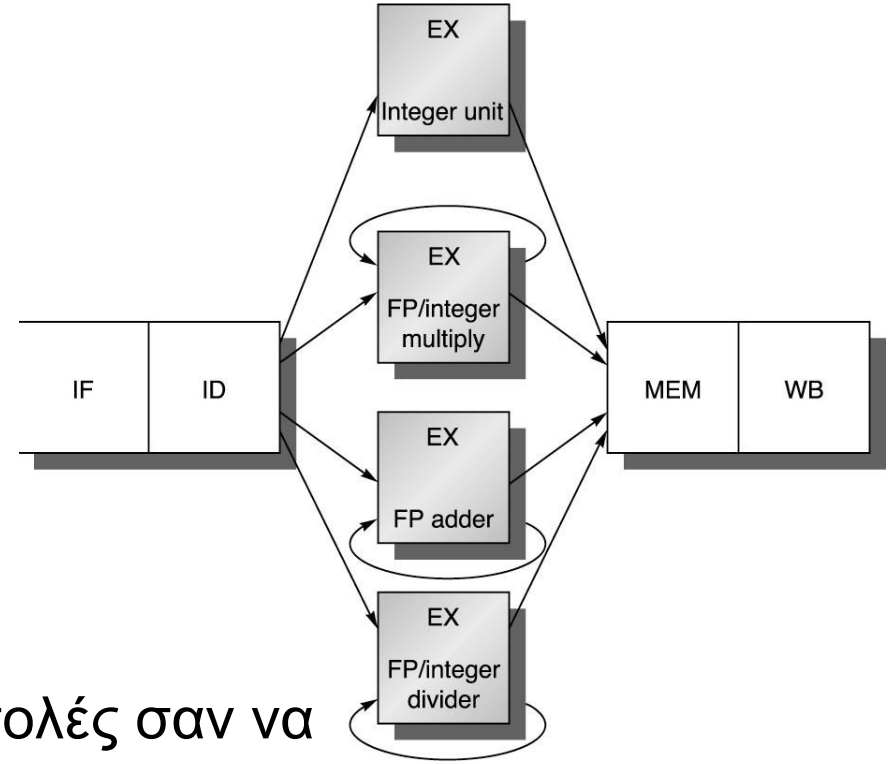


Λειτουργίες πολλαπλών κύκλων

- Στο κλασικό 5-stage pipeline όλες οι λειτουργίες χρειάζονται 1 κύκλο
- Το να έχουμε όλες τις εντολές να τελειώνουν σε έναν κύκλο σημαίνει:
  - μείωση της συχνότητας για να προσαρμοστεί το pipeline στη διάρκεια της πιο χρονοβόρας λειτουργίας , ή
  - χρησιμοποίηση εξαιρετικά πολύπλοκων κυκλωμάτων για την υλοποίηση της πιο χρονοβόρας λειτουργίας σε 1 κύκλο
- Ρεαλιστική αντιμετώπιση:
  - επέκταση του pipeline για να υποστηρίζει λειτουργίες διαφορετικής διάρκειας
- Παραδείγματα από πραγματικούς επεξεργαστές:
  - FP add, Int/FP mult: ~2-6 κύκλους
  - Int/FP div, sqrt: ~20-50 κύκλους
  - Προσπέλαση στη μνήμη: ~2-200 κύκλους...

Multi-cycle execution: επέκταση του 5-stage pipeline με non-pipelined FP units

- 4 διαφορετικές ALUs:
  - Integer
  - FP/Integer multiply
  - FP adder
  - FP/Integer divide



- φανταζόμαστε το EX για τις FP εντολές σαν να επαναλαμβάνεται για πολλούς συνεχόμενους κύκλους
- κάθε τέτοια μονάδα είναι non-pipelined: δε μπορεί να σταλεί ("issue") προς εκτέλεση σε μια μονάδα μια εντολή, αν κάποια προηγούμενη χρησιμοποιεί ακόμα τη μονάδα αυτή (structural hazard)
- η εντολή που stall-άρει καθυστερεί και όλες τις επόμενες

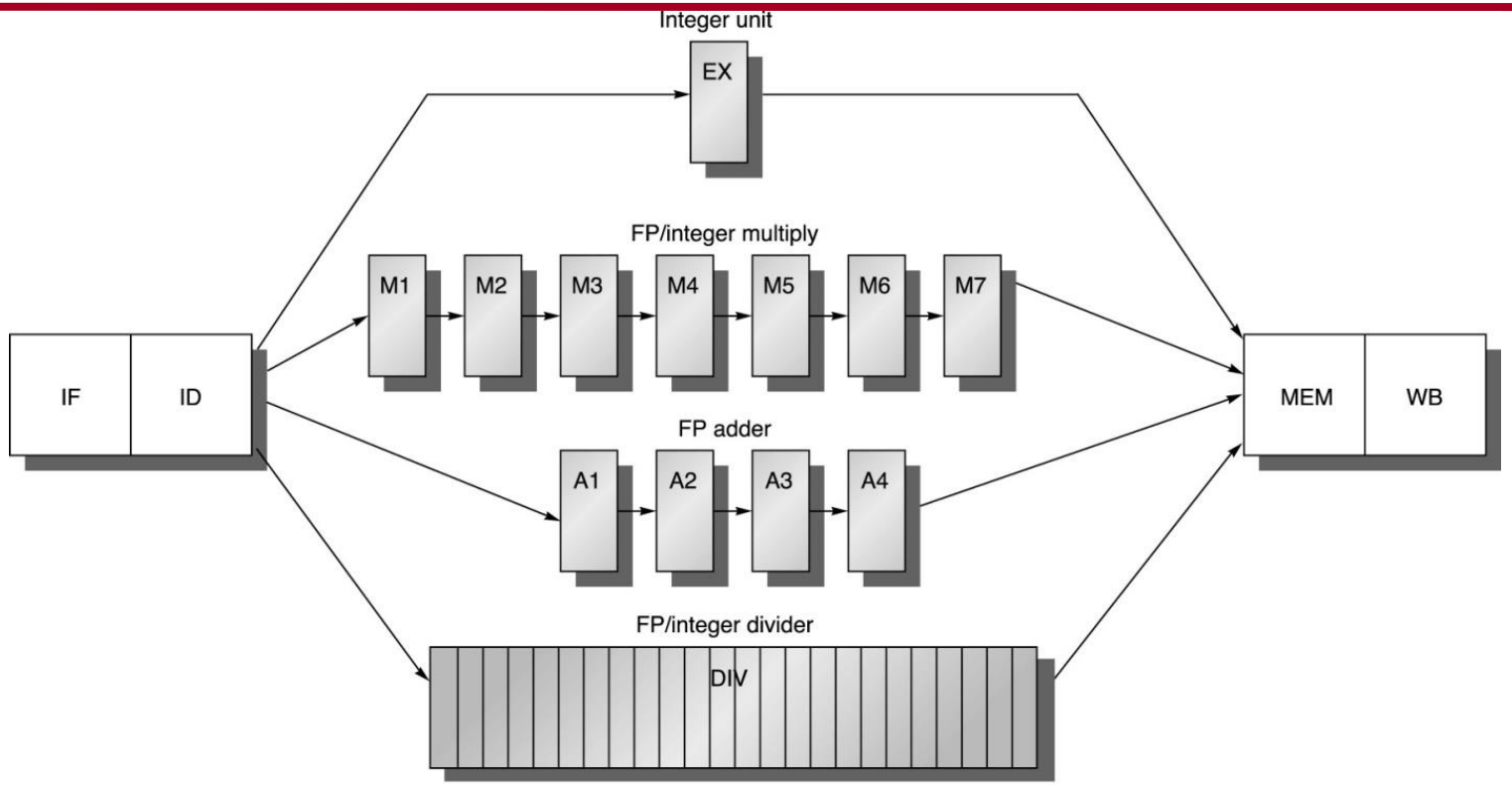
Μετρικές περιγραφής ενός multi-cycle pipeline

- Initiation interval: #κύκλων που μεσολαβούν ανάμεσα στην αποστολή στις μονάδες εκτέλεσης δύο εντολών ίδιου τύπου
  - ενδεικτικό του throughput μιας μονάδας εκτέλεσης
- Latency: #κύκλων που μεσολαβούν μεταξύ της εντολής που παράγει ένα αποτέλεσμα και της εντολής που θα το καταναλώσει

Unit	Latency	Initiation interval
Integer ALU	**0**	**1**
Data memory	**1**	**1**
FP add	**3**	**1**
FP multiply	**6**	**1**
FP divide	**24**	**25**

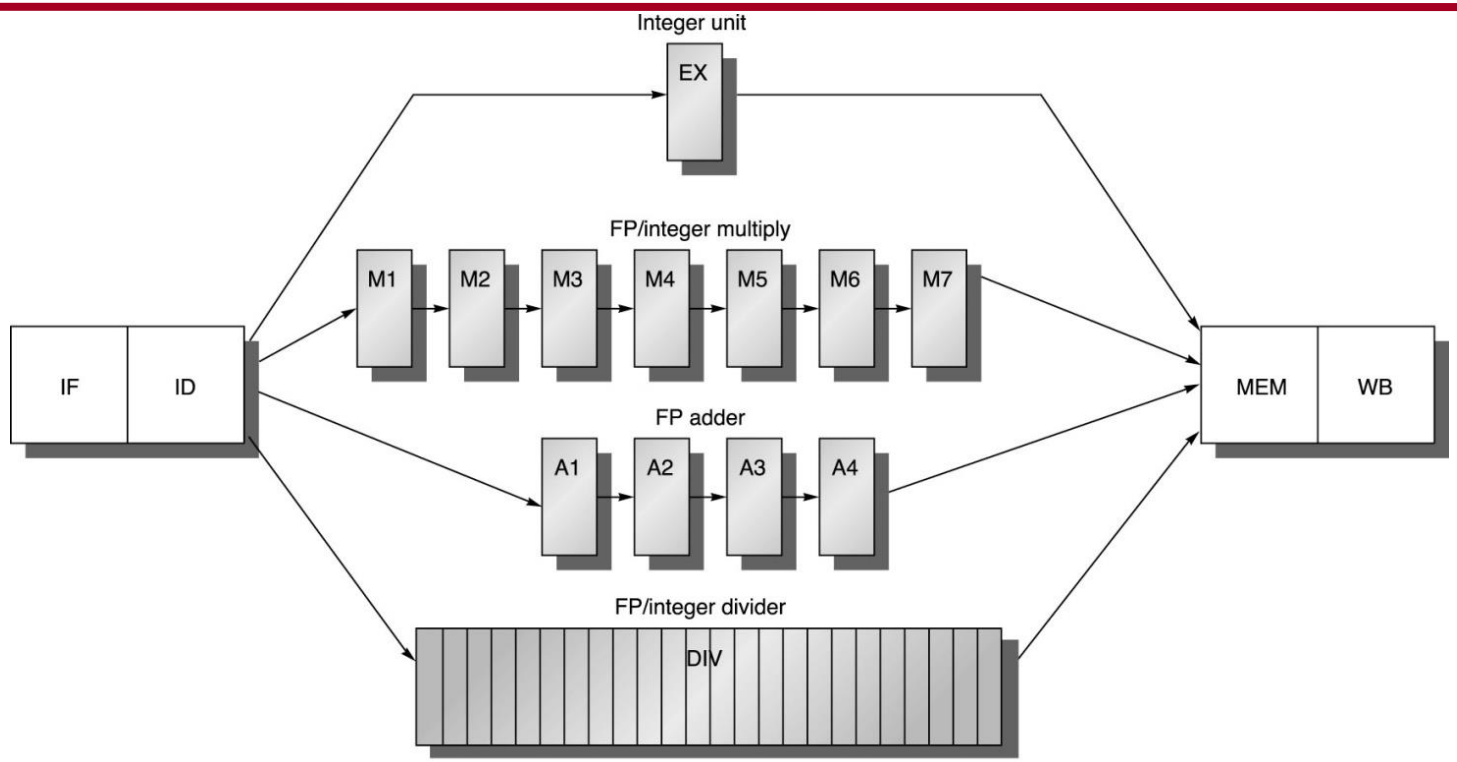
- Οι περισσότερες εντολές καταναλώνουν τους τελεστέους τους στην αρχή του EX
  - Latency = #σταδίων μετά το EX όπου μια εντολή παράγει το αποτέλεσμά της

Multi-cycle execution: επέκταση του 5-stage pipeline με pipelined FP units



- επιτρέπει να βρίσκονται εν εκτελέσει μέχρι 4 FP-adds, 7 FP-muls, 1 FP-divide (non-pipelined)
- τα επιμέρους στάδια είναι ανεξάρτητα και χωρίζονται με ενδιάμεσους καταχωρητές
- διάσπαση μιας λειτουργίας σε πολλά επιμέρους στάδια:
  - ↑ συχνότητα ρολογιού
  - ↑ latency λειτουργιών + ↑ συχνότητα RAW hazards + ↑ stalls

Παράδειγμα



Clock Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
MUL.D	IF	ID	M1	M2	M3	M4	M5	M6	M7	M	WB							
ADD.D		IF	ID	A1	A2	A3	A4	M	WB									
L.D			IF	ID	EX	M	WB											
S.D				IF	ID	EX	M	WB										

Hazards

- Νέα ζητήματα που προκύπτουν:
  - η μονάδα divide δεν υποστηρίζει pipelining → μπορεί να προκύψουν structural hazards
  - εντολές διαφορετικού latency → #εγγραφών στο register file σε έναν κύκλο μπορεί να είναι >1 (structural hazards)
  - οι εντολές μπορεί να φτάσουν στο WB εκτός σειράς προγράμματος → μπορούν να συμβούν WAW hazards (WAR?)
  - οι εντολές μπορεί να ολοκληρωθούν εκτός σειράς προγράμματος → πρόβλημα με τις εξαιρέσεις
  - μεγαλύτερο latency στις εντολές → συχνότερα τα stalls εξαιτίας RAW hazards

RAW hazards και αύξηση των stalls

	Clock Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
L.D F4,0(R2)		IF	ID	EX	M	WB													
MUL.D F0,F4,F6			IF	ID	S	M1	M2	M3	M4	M5	M6	M7	M	WB					
ADD.D F2,F0,F8				IF	S	ID	S	S	S	S	S	S	A1	A2	A3	A4	M	WB	
S.D F2,0(R2)						IF	S	S	S	S	S	S	ID	EX	S	S	S	M	WB

- full bypassing/forwarding
- η S.D πρέπει να καθυστερήσει έναν κύκλο παραπάνω για να αποφύγουμε το conflict στο MEM της ADD.D

Structural Hazards

Clock Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
MUL.D F0,F4,F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	M	WB				
…		IF	ID	EX	M	WB									
…			IF	ID	EX	M	WB								
ADD.D F2,F4,F6				IF	ID	A1	A2	A3	A4	M	WB				
…					IF	ID	EX	M	WB						
…						IF	ID	EX	M	WB					
L.D F2,0(R2)							IF	ID	EX	M	WB				

Με μόνο ένα write port στο register file → structural hazard

- multiple write ports (…όμως δεν είναι η συνήθης περίπτωση στα προγράμματα)
- interlocks:
  - ελέγχουμε στο ID το ενδεχόμενο η τρέχουσα εντολή να έχει conflict στο WB με μία προηγούμενη εντολή, και αν ισχύει αυτό την stall-άρουμε στο ID (προτού γίνει issue)
  - ελέγχουμε το ενδεχόμενο η εντολή να έχει conflict στο WB όταν αυτή πάει να μπει στο MEM ή στο WB, και αν ισχύει αυτό τη stall-άρουμε εκεί

WAW Hazards

Clock Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
...		IF	ID	EX	M	WB									
...			IF	ID	EX	M	WB								
ADD.D F2,F4,F6				IF	ID	A1	A2	A3	A4	M	WB				
inst					IF	ID	EX	M	WB						
L.D F2,0(R2)						IF	ID	EX	M	WB					

- Μπορεί να προκύψει μόνο αν η “inst” δε χρησιμοποιεί τον F2
  - ποιος ο λόγος να έχουμε δύο producers του ίδιου πράγματος χωρίς ενδιάμεσο consumer???
  - σπάνια περίπτωση, αφού ένας «λογικός» compiler θα έκανε eliminate τον πρώτο producer
  - μπορεί όμως να συμβεί!! π.χ. Loops, όταν η σειρά εκτέλεσης των εντολών δεν είναι η αναμενόμενη (branch delay slots, εντολές σε trap handlers, κ.λπ.),κ.α.
- Το hazard ανιχνεύεται στο ID, όταν η L.D είναι έτοιμη να γίνει issue
- Αντιμετώπιση:
  - καθυστερούμε το issue της L.D μέχρι η ADD.D να μπει στο MEM
  - απορρίπτουμε την εγγραφή της ADD.D → η L.D μπορεί να γίνει issue άμεσα
- Η δυσκολία δεν έγκειται τόσο στην αντιμετώπιση του hazard, *όσο στο να βρούμε ότι η L.D μπορεί να ολοκληρωθεί πιο γρήγορα από την ADD.D!*

Συνοψίζοντας

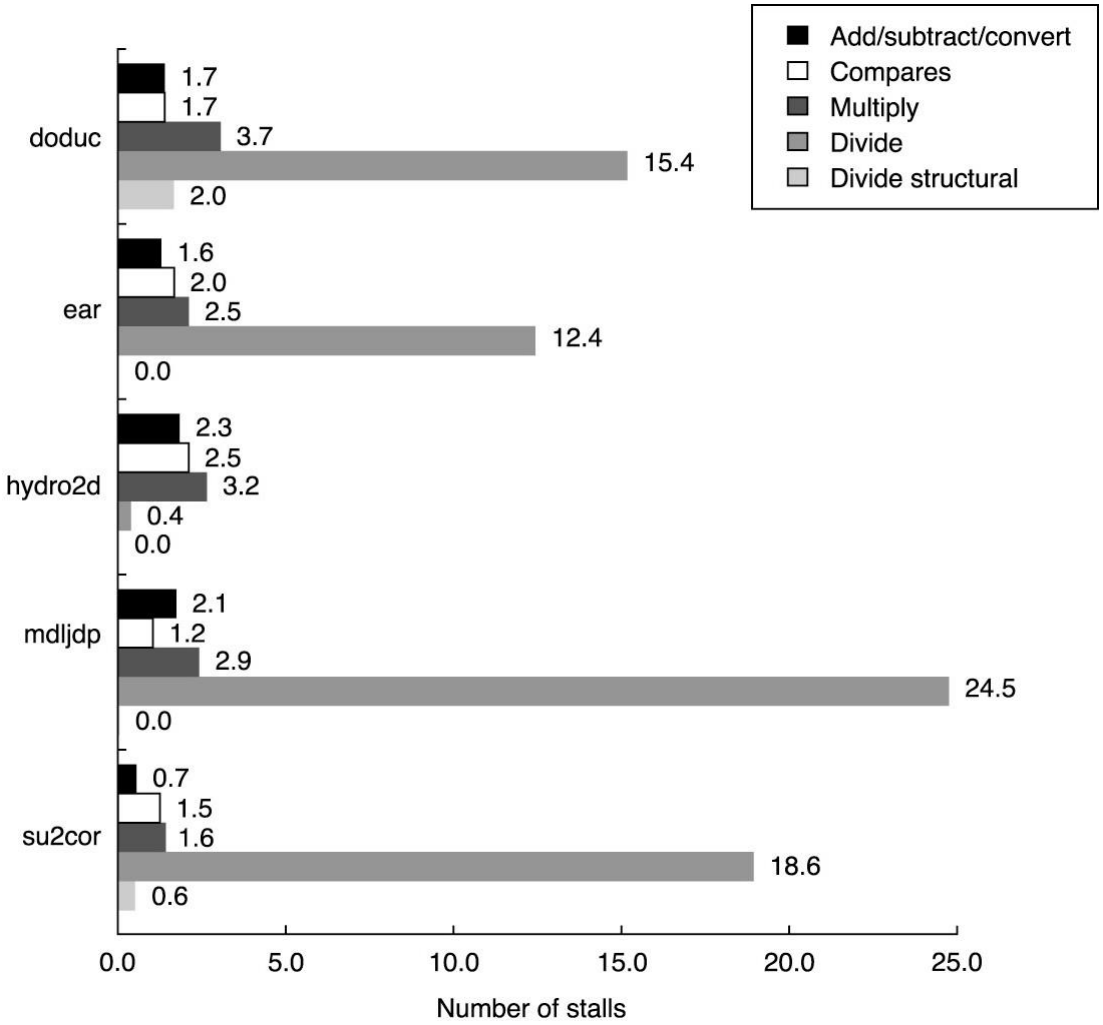
- Τρεις επιπλέον έλεγχοι για hazards στο ID προτού μια εντολή γίνει issue:
  - structural:
    - εξασφάλισε ότι η (non-pipelined) μονάδα δεν είναι απασχολημένη
    - εξασφάλισε ότι το write port του register file θα είναι διαθέσιμο όταν θα ζητηθεί
  - RAW:
    - περίμενε μέχρι οι source registers της issuing εντολής να μην υπάρχουν πλέον σαν destinations στους ενδιάμεσους pipeline registers των pipelined μονάδων εκτέλεσης
    - π.χ. για τη μονάδα FP-add: αν η εντολή στο ID έχει σαν source τον F2, τότε για να μπορεί να γίνει issue, ο F2 δε θα πρέπει να συγκαταλέγεται στα destinations των ID/A1, A1/A2, A2/A3.
  - WAW:
    - έλεγξε αν κάποια εντολή στα στάδια A1,…,A4,D,M1,…,M7 έχει τον ίδιο destination register με αυτή στο ID, και αν ναι, stall-αρε την τελευταία στο ID
- Προώθηση:
  - έλεγξε αν το destination κάποιου από τους EX/MEM, A4/MEM, M7/MEM, D/MEM, MEM/WB ταυτίζεται με τον source register κάποιας FP εντολής, και αν ναι, ενεργοποίησε τον κατάλληλο πολυπλέκτη

Απόδοση του multi-cycle FP pipeline

- stall cycles ανά FP-εντολή

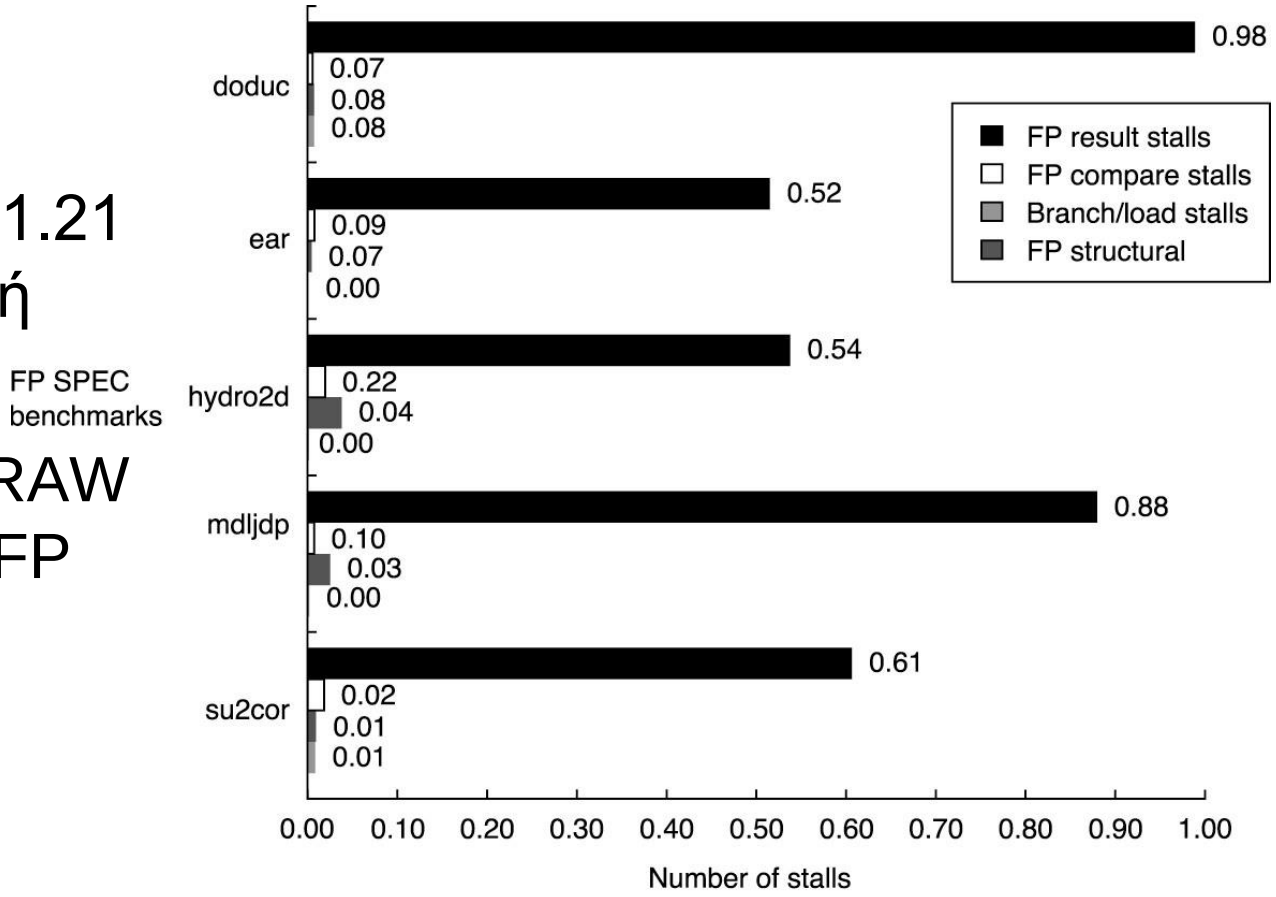
- τα RAW hazard stalls «ακολουθούν» το latency της αντίστοιχης μονάδας, π.χ.:
  - μέσος #stalls που οφείλεται στην MUL.D=2.8 (46% του latency της FP-Mult)
  - μέσος #stalls που οφείλονται στην DIV.D=14.2 (59% του latency της FP-Div)

- τα structural hazards είναι σπάνια, επειδή τα divides δεν είναι συχνά

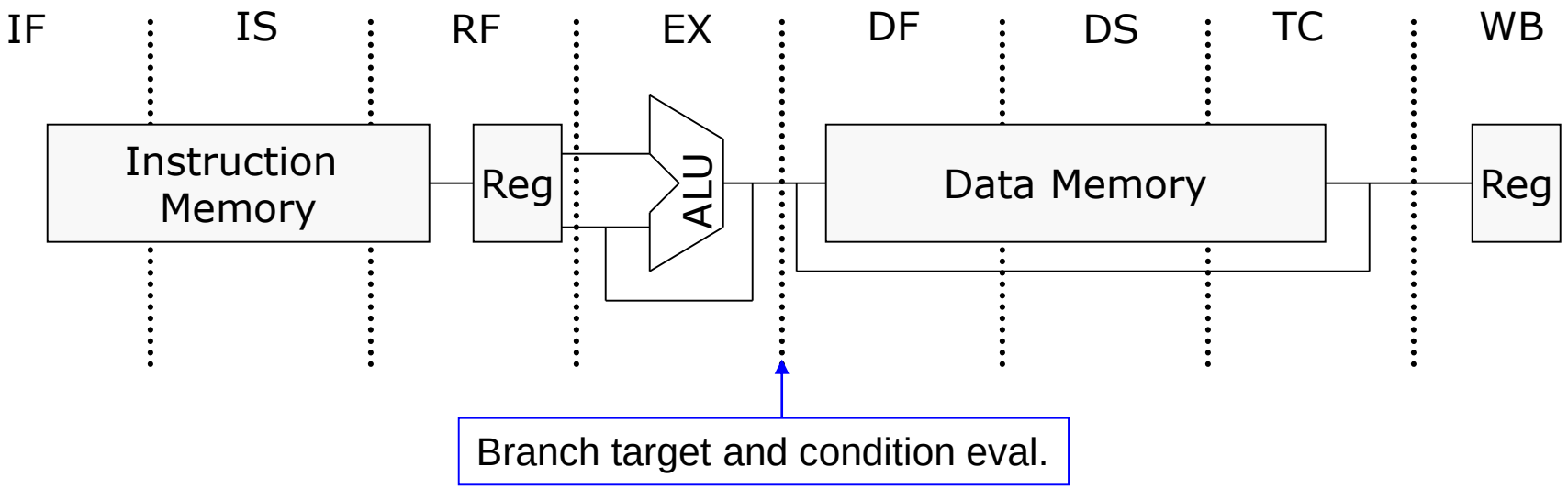


Απόδοση του multi-cycle FP pipeline

- stalls ανά εντολή + breakdown
- από 0.65 μέχρι 1.21 stalls ανά εντολή
- κυριαρχούν τα RAW hazard stalls («FP result stalls»)

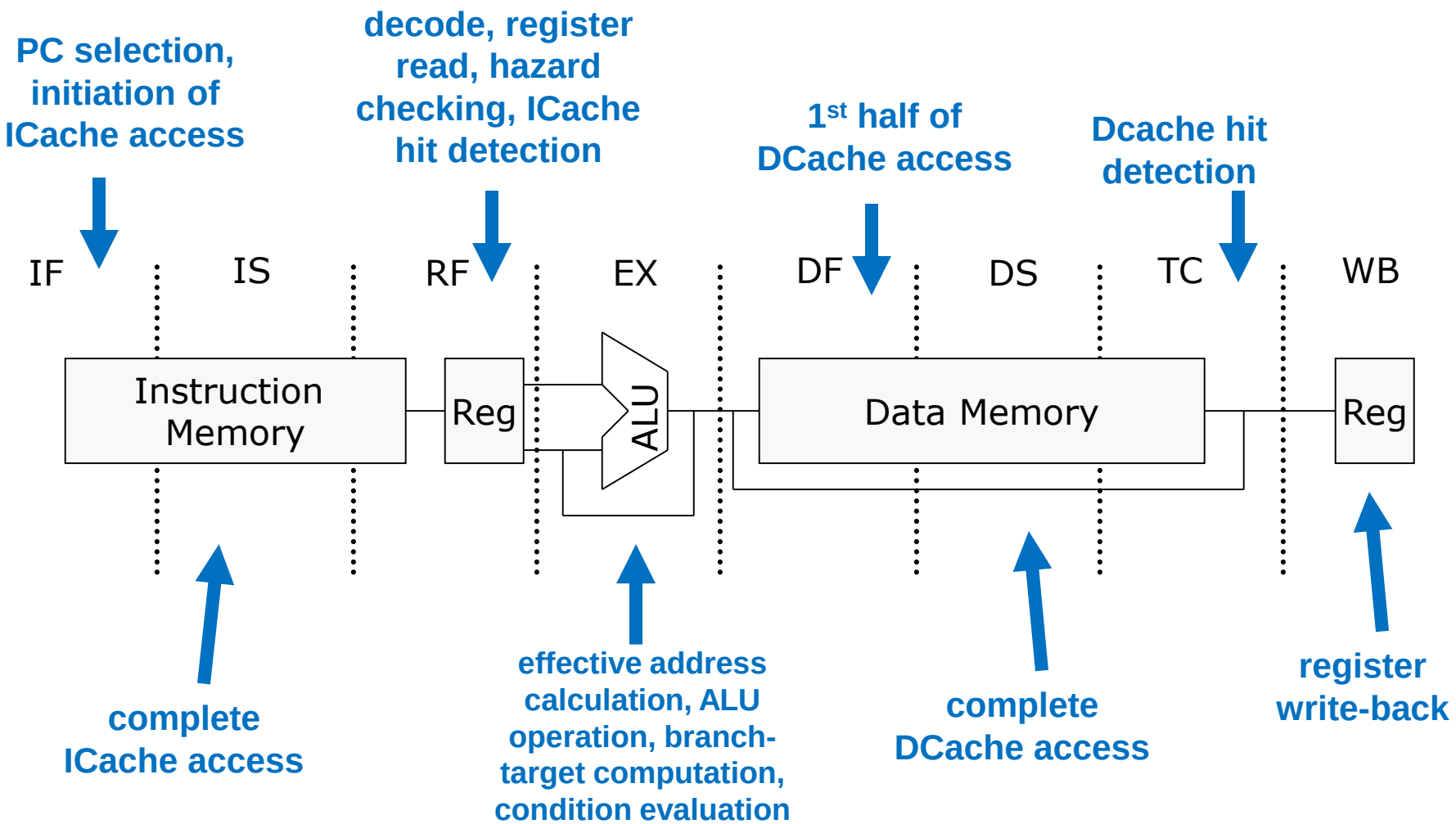


Case-study: MIPS R4000 Pipeline

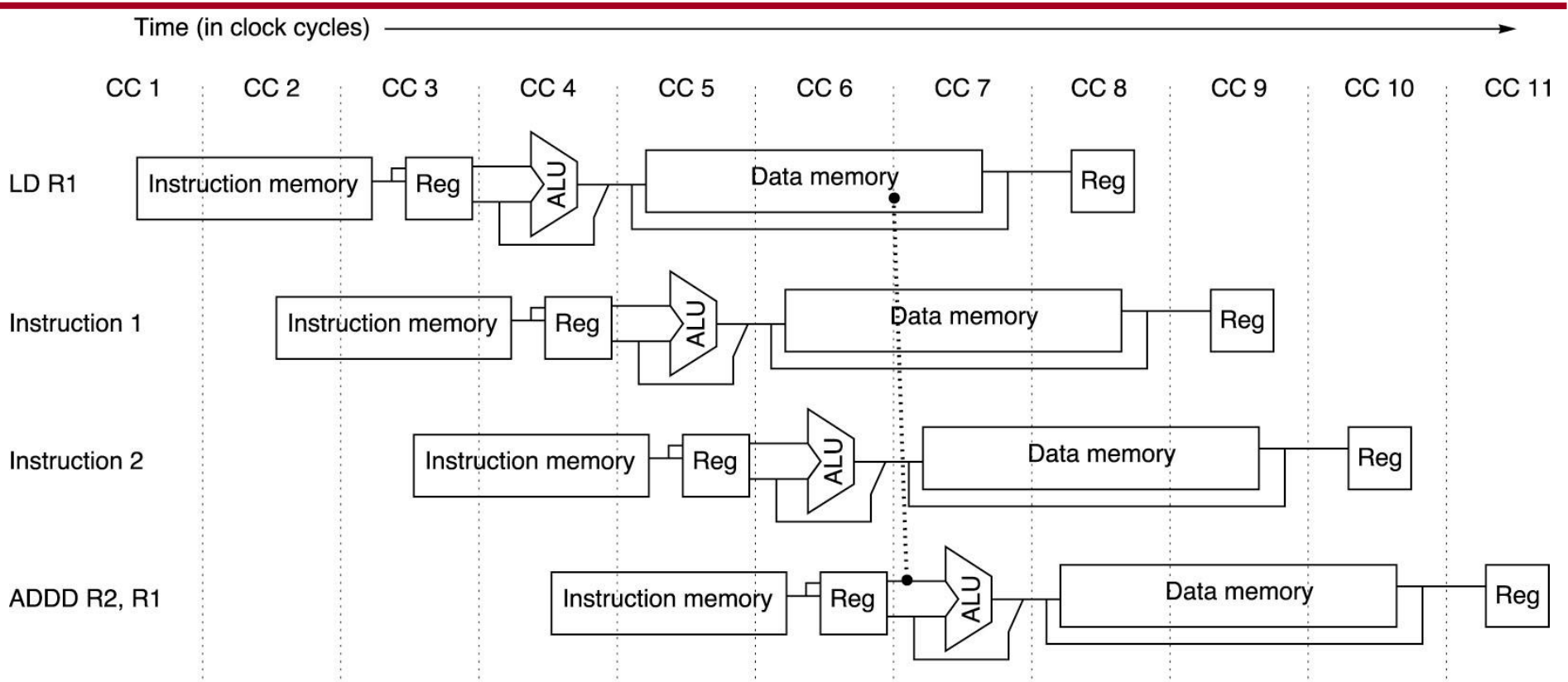


- Deeper Pipeline (superpipelining): επιτρέπει υψηλότερα clock rates
- Fully pipelined memory accesses (2 cycle delays για loads)
- Predicted-Not-Taken πολιτική
  - Not-taken (fall-through) branch : 1 delay slot
  - Taken branch: 1 delay slot + 2 idle cycles

Case-study: MIPS R4000 Pipeline



Load delay (2 cycles)



- στην πραγματικότητα, το pipeline μπορεί να προωθήσει τα δεδομένα από την cache πριν διαπιστώσει αν είναι hit ή miss!

Branch delay (3 cycles)