Thread Level Parallelism
&
Data Level Parallelism

Ο "Νόμος" της απόδοσης των μικροεπεξεργαστών

$$\frac{1}{\text{Performance}} = \frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

(instr. count) (CPI) (cycle time)

$$\rightarrow \text{Performance} = \frac{\text{IPC x Hz}}{\text{instr. count}}$$

Αύξηση απόδοσης

- clock speed (↑Hz)
- αρχιτεκτονικές βελτιστοποιήσεις (↑IPC):
  - pipelining, superscalar execution, branch prediction, out-of-order execution, caches
- αλγόριθμοι (↓instr.count)

Περιορισμοί Αύξησης Απόδοσης

- Ο ILP εκμεταλλεύεται παράλληλες λειτουργίες, συνήθως μη οριζόμενες από τον προγραμματιστή
  - σε μια «ευθεία» ακολουθία εντολών (χωρίς branches)
  - ανάμεσα σε διαδοχικές επαναλήψεις ενός loop
- Δύσκολο το να εξάγουμε ολοένα και περισσότερο ILP από ένα και μόνο νήμα εκτέλεσης
  - εγγενώς χαμηλός ILP σε πολλές εφαρμογές

  → ανεκμετάλλευτες πολλές από τις μονάδες ενός superscalar επεξεργαστή
- Συχνότητα ρολογιού: φυσικά εμπόδια στη συνεχόμενη αύξησή της
  - μεγάλη έκλυση θερμότητας, μεγάλη κατανάλωση ισχύος, διαρροή ρεύματος

Πρέπει να βρούμε άλλον τρόπο πέρα από τον ILP + συχνότητα ρολογιού

για να βελτιώσουμε την απόδοση

Προσεγγίσεις Αύξησης Απόδοσης

- **Παραλληλισμός σε επίπεδο εντολής (Instruction Level Parallelism – ILP)**
  - Εξαρτάται από τις πραγματικές εξαρτήσεις δεδομένων που υφίστανται ανάμεσα στις εντολές.

- **Παραλληλισμός σε επίπεδο νήματος (Thread-Level Parallelism – TLP)**
  - Αναπαρίσταται ρητά από τον προγραμματιστή, χρησιμοποιώντας πολλαπλά νήματα εκτέλεσης τα οποία είναι εκ κατασκευής παράλληλα.

- **Παραλληλισμός σε επίπεδο δεδομένων (Data-Level Parallelism – DLP)**
  - Αναπαρίσταται ρητά από τον προγραμματιστή ή δημιουργείται αυτόματα από τον μεταγλωττιστή.
  - Οι ίδιες εντολές επεξεργάζονται πολλαπλά δεδομένα ταυτόχρονα

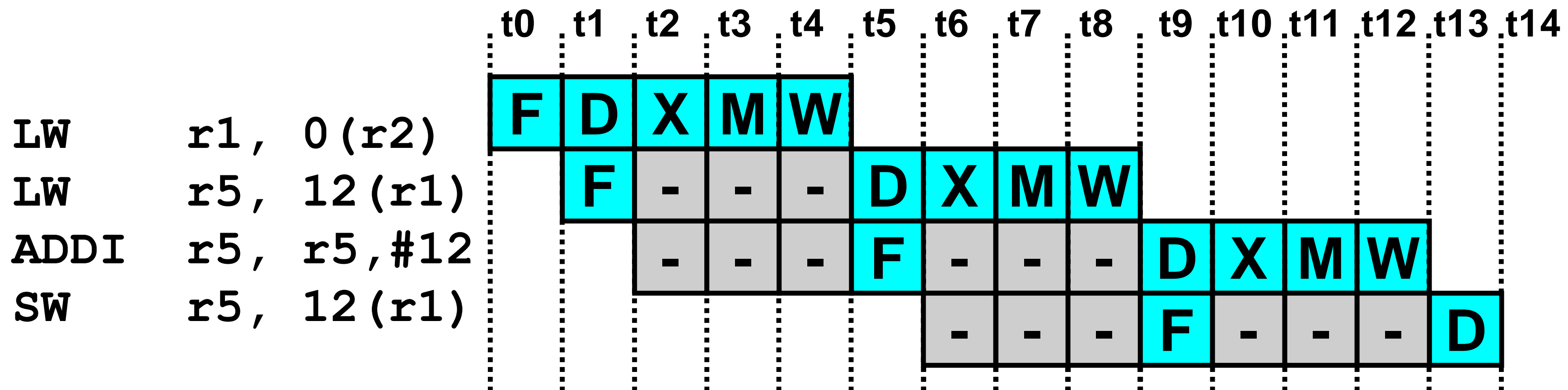
Πολυνηματικές Αρχιτεκτονικές

Πολυνηματισμός

- Σκοπός: χρήση πολλών ανεξάρτητων instruction streams από πολλαπλά νήματα εκτέλεσης
- **Παραλληλισμός σε επίπεδο νήματος (Thread-Level Parallelism – TLP)** Αναπαρίσταται ρητά από τον προγραμματιστή, χρησιμοποιώντας πολλαπλά νήματα εκτέλεσης τα οποία είναι εκ κατασκευής παράλληλα

- Πολλά φορτία εργασίας έχουν σαν χαρακτηριστικό τους τον TLP:
  - TLP σε πολυπρογραμματιζόμενα φορτία (εκτέλεση ανεξάρτητων σειριακών εφαρμογών)
  - TLP σε πολυνηματικές εφαρμογές (επιτάχυνση μιας εφαρμογής διαχωρίζοντάς την σε νήματα και εκτελώντας τα παράλληλα)
- Οι πολυνηματικές αρχιτεκτονικές χρησιμοποιούν τον TLP σε τέτοια φορτία εργασίας για να βελτιώσουν τα επίπεδα χρησιμοποίησης των μονάδων του επεξεργαστή
  - βελτίωση του throughput πολυπρογραμματιζόμενων φορτίων
  - βελτίωση του χρόνου εκτέλεσης πολυνηματικών εφαρμογών

Παράδειγμα: κίνδυνοι δεδομένων



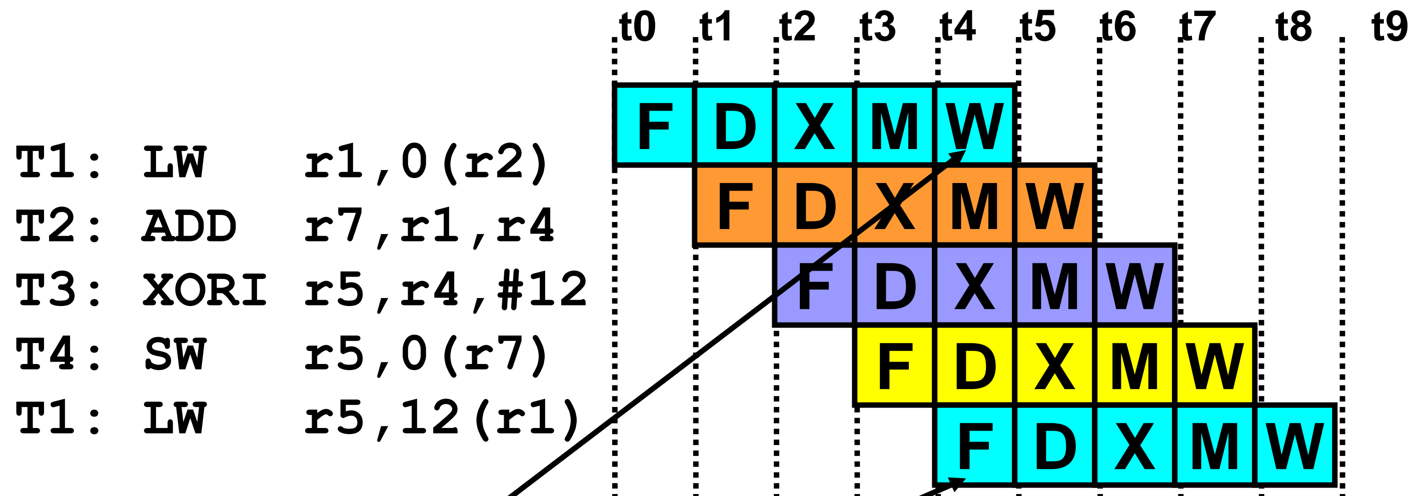
- Οι εξαρτήσεις μεταξύ των εντολών αποτελούν περιοριστικό παράγοντα για την εξαγωγή παραλληλισμού
- Τι μπορεί να γίνει προς αυτή τη κατεύθυνση;

Αντιμετώπιση με πολυνηματισμό

Πώς μπορούμε να μειώσουμε τις εξαρτήσεις μεταξύ των εντολών σε ένα pipeline?

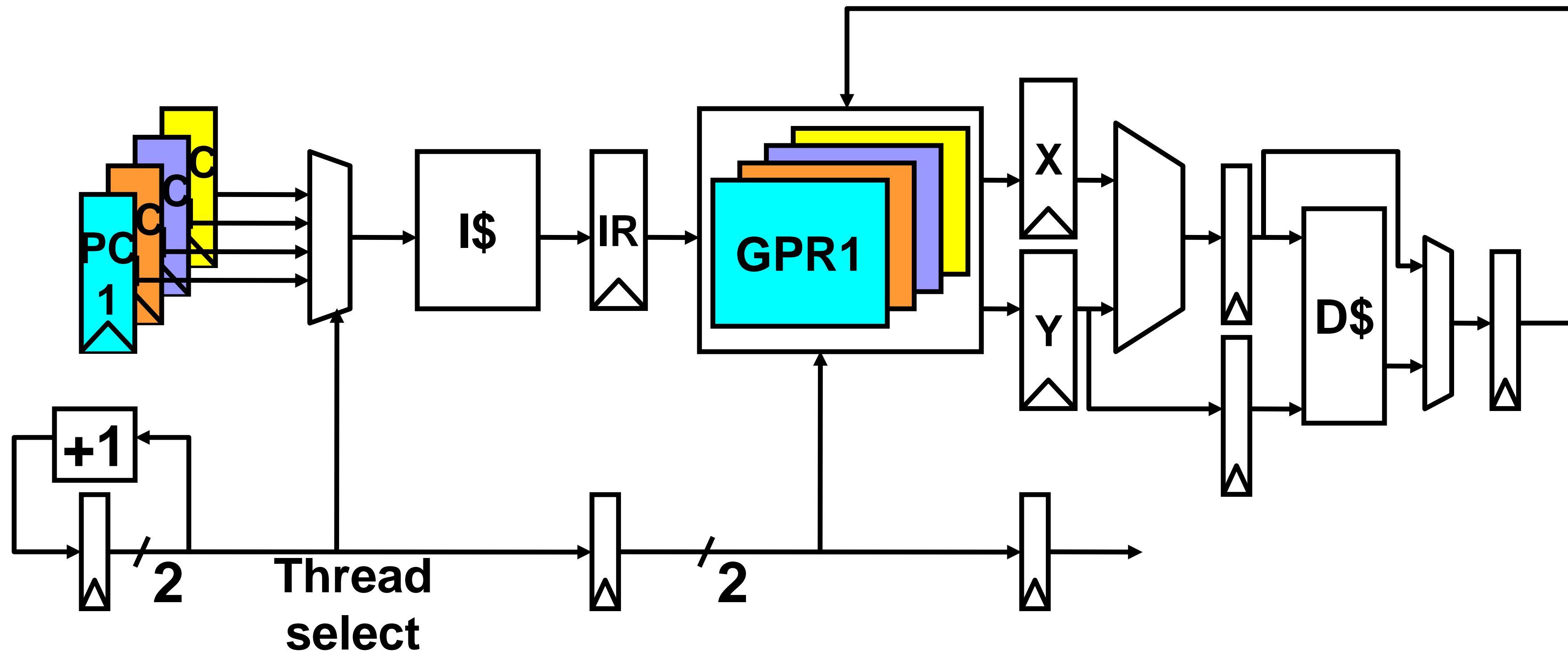
- Ένας τρόπος είναι να επικαλύψουμε την εκτέλεση εντολών από διαφορετικά νήματα στο ίδιο pipeline…

Επικάλυψη εκτέλεσης 4 νημάτων, T1-T4, στο απλό 5-stage pipeline



Η προηγούμενη εντολή στο νήμα ολοκληρώνει το WB προτού η επόμενη εντολή στο ίδιο νήμα διαβάσει το register file

Απλή μορφή πολυνηματικού pipeline



- Το software «βλέπει» πολλαπλές CPUs
- Κάθε νήμα χρειάζεται να διατηρεί τη δική του αρχιτεκτονική κατάσταση
  - program counter
  - general purpose registers
- Τα νήματα μοιράζονται τις ίδιες μονάδες εκτέλεσης
- Hardware για γρήγορη εναλλαγή των threads
  - πρέπει να είναι πολύ πιο γρήγορη από ένα software-based process switch (≈ 100s - 1000s κύκλων)

Πολυνηματισμός

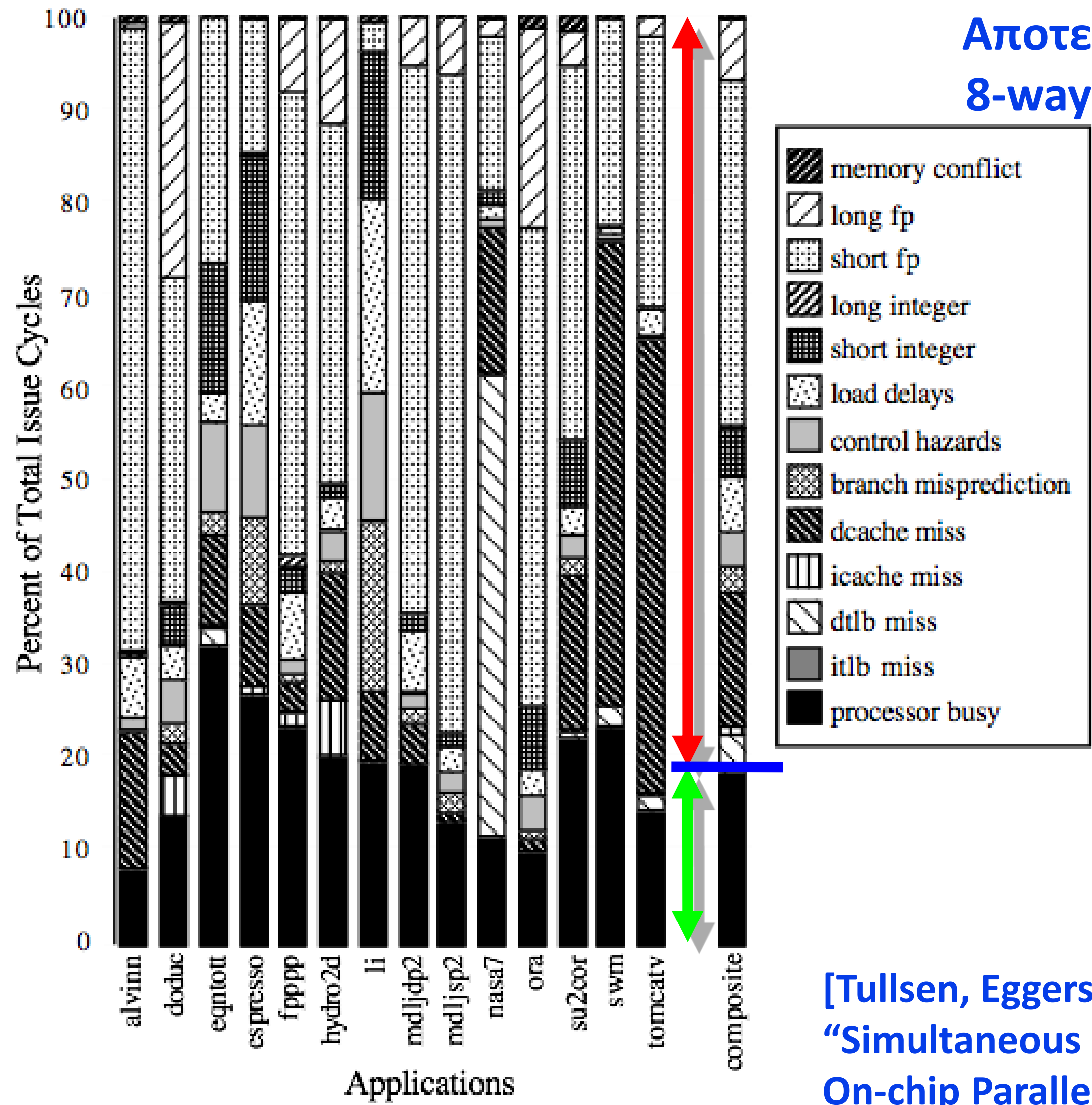
- Πλεονεκτήματα
  - Δε χρειάζεται dependency checking μεταξύ των εντολών
  - Δε χρειάζεται branch prediction
  - Αποφυγή bubbles πραγματοποιώντας χρήσιμη δουλειά από άλλα threads
  - Βελτίωση system throughput, utilization, latency tolerance

- Μειονεκτήματα
  - Πολυπλοκότητα hardware (PCs, register files, thread selection logic, …)
  - Χειρότερο single thread performance (1 instruction every N cycles)
  - Υψηλός ανταγωνισμός για πόρους (resource contention for caches & memory)

Επιπλέον υλικό και πληροφορίες:

- Mario Nemirovsky, Dean M. Tullsen. **Multithreading Architecture**. Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers 2013, ISBN 9781608458554.

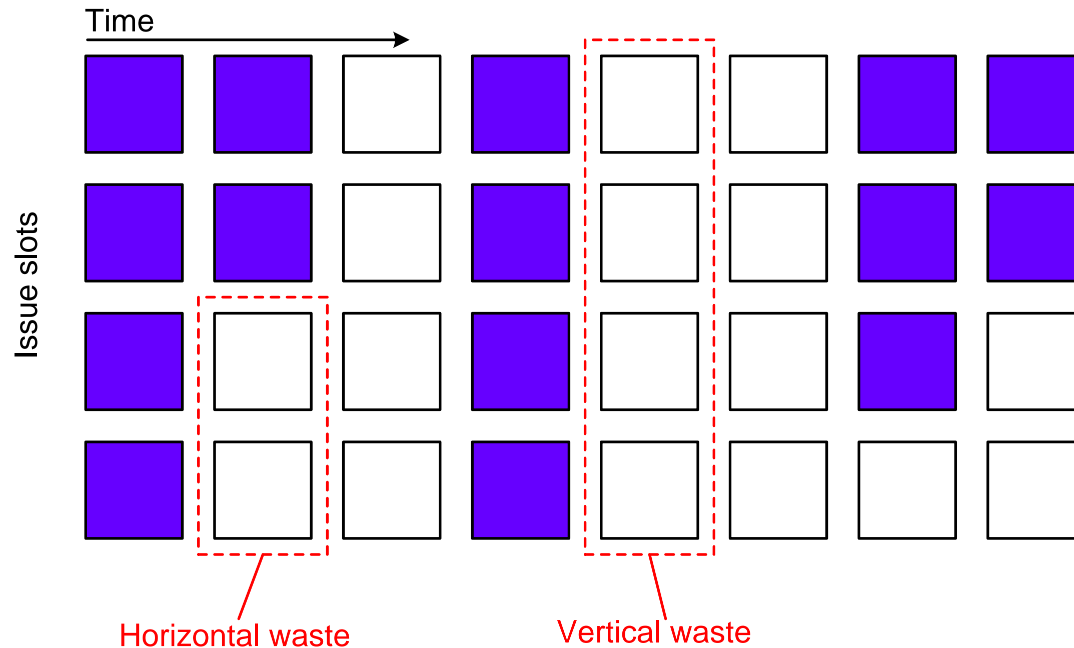
Για αρκετές εφαρμογές, οι περισσότερες μονάδες εκτέλεσης σε έναν OoO superscalar μένουν ανεκμετάλλευτες



Αποτελέσματα για 8-way superscalar.

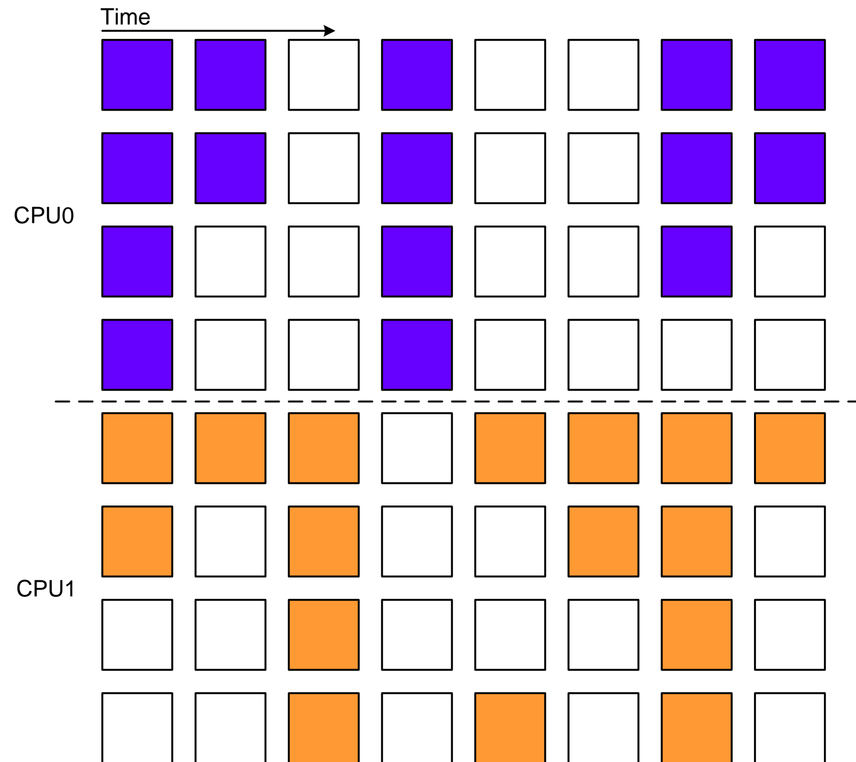
[Tullsen, Eggers, and Levy, "Simultaneous Multithreading: Maximizing On-chip Parallelism, ISCA 1995.]

OoO superscalar



- Horizontal waste: εξαιτίας χαμηλού ILP
- Vertical waste: εξαιτίας long-latency γεγονότων
  - cache misses
  - pipeline flushes λόγω branch mispredictions

Chip Multi-Processor

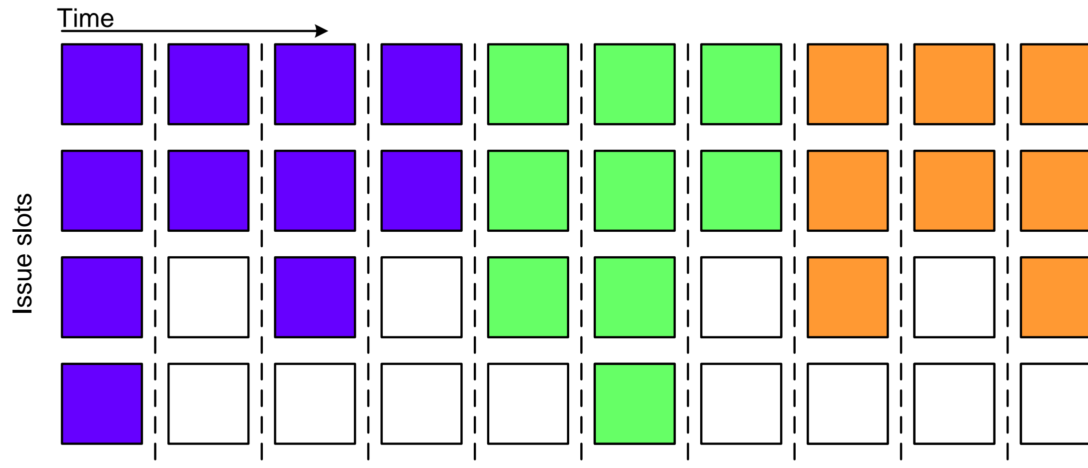


- τα προβλήματα εξακολουθούν να υφίστανται…

Υλοποιήσεις Πολυνηματισμού

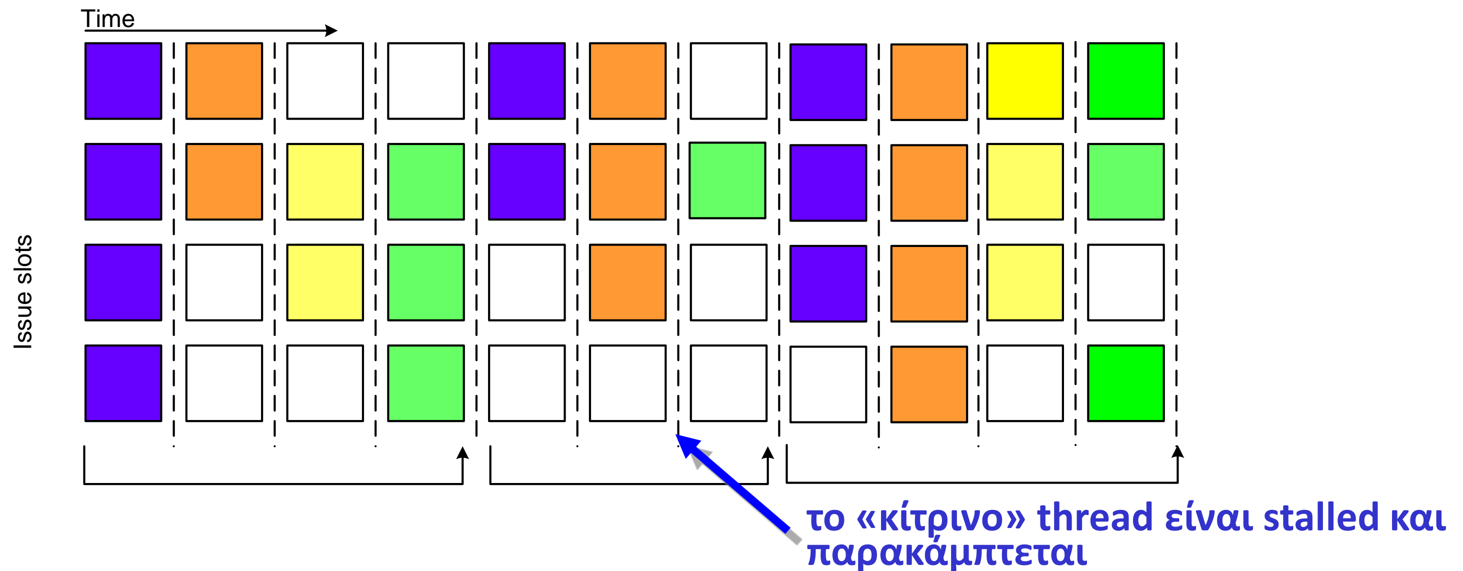
1. Coarse-grained multithreading
2. Fine-grained multithreading
3. Simultaneous multithreading (SMT)

Coarse-grained multithreading (“switch-on-event”)



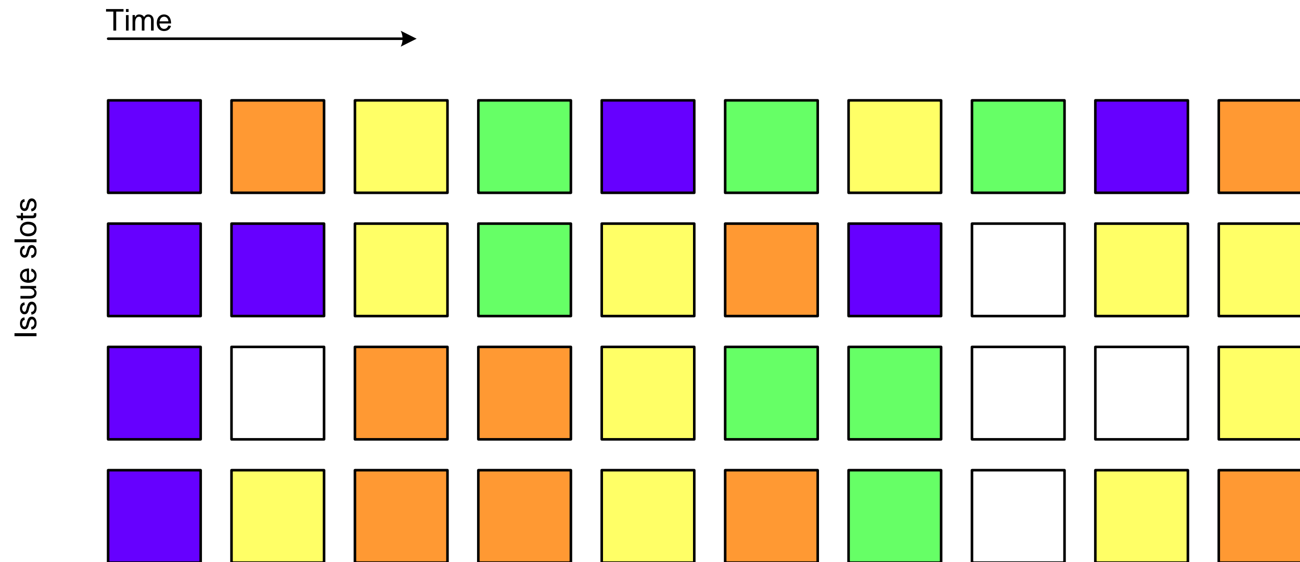
- Εναλλαγή thread μόνο μετά από stall του thread που εκτελείται, π.χ. λόγω L2 cache miss
- Πλεονεκτήματα:
  - δε χρειάζεται να έχει πολύ γρήγορο μηχανισμό εναλλαγής των threads
  - δεν καθυστερεί την εκτέλεση ενός thread, αφού οι εντολές από άλλα threads γίνονται issue μόνο όταν το thread αντιμετωπίσει κάποιο stall
- Μειονεκτήματα:
  - δεν αντιμετωπίζει το horizontal waste
  - σε μικρά stalls, η εναλλαγή του stalled thread και η δρομολόγηση στο pipeline κάποιου έτοιμου thread μπορεί να έχει απώλειες στην απόδοση του πρώτου thread αν τελικά οι κύκλοι που stall-άρει είναι λιγότεροι από τους κόστος εκκίνησης του pipeline με το νέο thread
- Εξαιτίας αυτού του start-up κόστους, το coarse-grained multithreading είναι καλύτερο για την μείωση του κόστους από *μεγάλα stalls*, για τα οποία το stall time >> pipeline refill time
- e.g., IBM AS/400, DYSEAC, TX-2

Fine-grained multithreading



- Εναλλαγή μεταξύ των threads σε κάθε κύκλο, με αποτέλεσμα την επικάλυψη της εκτέλεσης των threads
  - η CPU είναι αυτή που κάνει την εναλλαγή σε κάθε κύκλο
- Γίνεται με round-robin τρόπο (κυκλικά), παρακάμπτοντας threads τα οποία είναι stalled σε κάποιο long-latency γεγονός
- Αντιμετωπίζει το vertical waste, τόσο για μικρά όσο και για μεγάλα stalls, αφού όταν ένα thread είναι stalled το επόμενο μπορεί να γίνει issue
- Μειονεκτήματα:
  - δεν αντιμετωπίζει το horizontal waste
  - καθυστερεί την εκτέλεση ενός thread το οποίο είναι έτοιμο να εκτελεστεί, χωρίς stalls, αφού ανάμεσα σε διαδοχικούς κύκλους αυτού του thread παρεμβάλλονται κύκλοι εκτέλεσης από όλα τα υπόλοιπα threads
- e.g., UltraSPARC T1 (“Niagara”), Cray MTA, CDC 6600, Delco TIO

Simultaneous Multithreading (SMT)

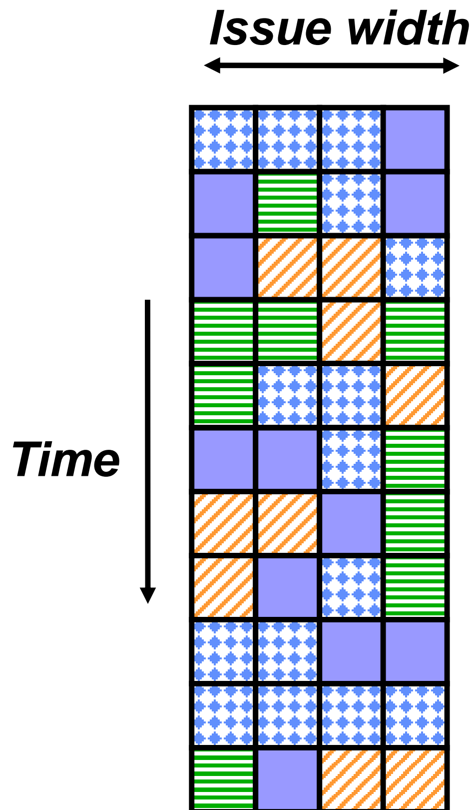


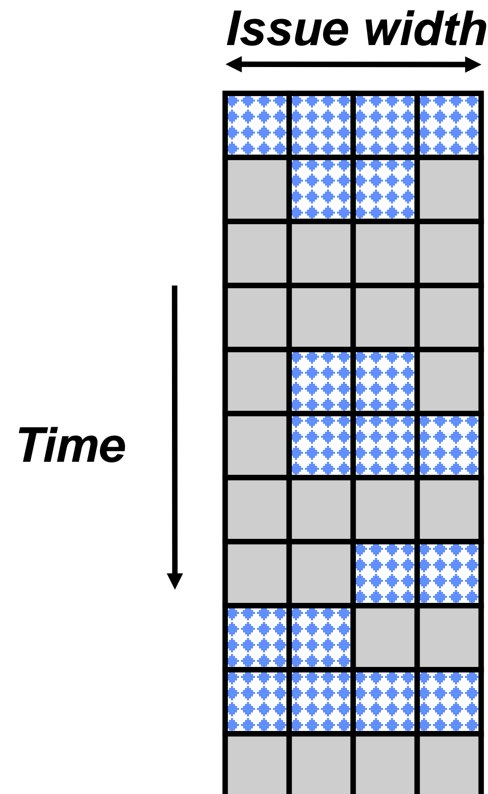
- Γίνονται issue εντολές από πολλαπλά νήματα ταυτόχρονα
  - αντιμετωπίζεται το horizontal waste
- Όταν ένα νήμα stall-άρει λόγω ενός long-latency γεγονότος, τα υπόλοιπα νήματα μπορούν να δρομολογηθούν και να χρησιμοποιήσουν τις διαθέσιμες μονάδες εκτέλεσης
  - αντιμετωπίζεται το vertical waste
- Μέγιστη χρησιμοποίηση των επεξεργαστικών πόρων από ανεξάρτητες λειτουργίες

Προσαρμοστικότητα του SMT στο είδος του διαθέσιμου παραλληλισμού

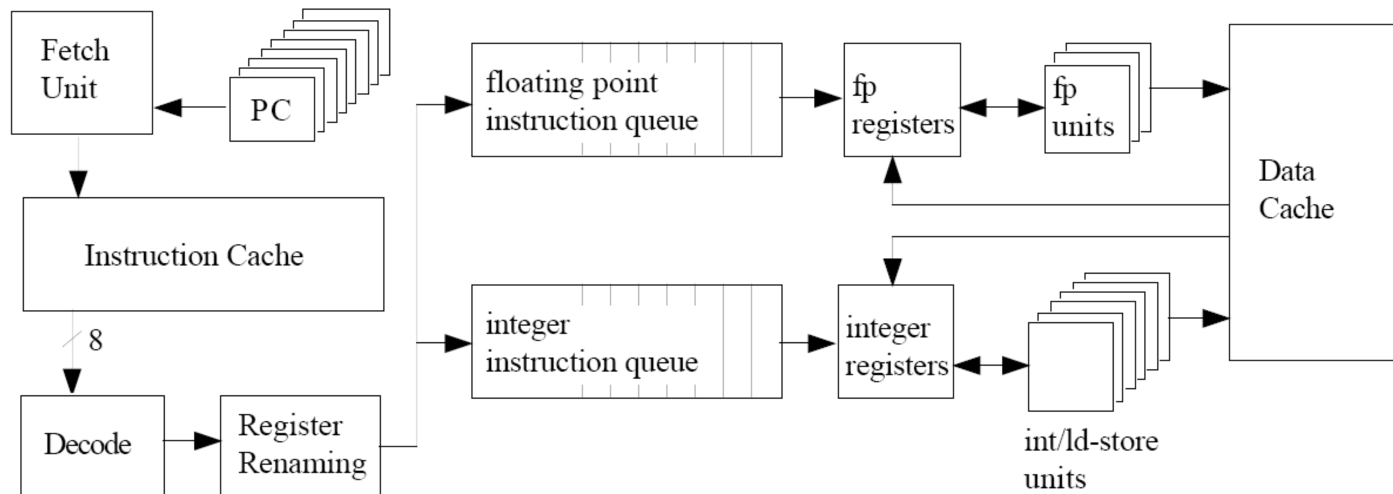
Για περιοχές με υψηλά επίπεδα TLP, ολόκληρο το έυρος του επεξεργαστή μοιράζεται από όλα τα threads

Για περιοχές με χαμηλά επίπεδα TLP, ολόκληρο το εύρος του επεξεργαστή είναι διαθέσιμο για την εκμετάλλευση του (όποιου) ILP





Αρχιτεκτονική SMT



- Βασικές επεκτάσεις σε σχέση με μια συμβατική superscalar αρχιτεκτονική
  - **πολλαπλοί program counters** και κατάλληλος μηχανισμός μέσω του οποίου η fetch unit επιλέγει κάποιον από αυτούς σε κάθε κύκλο (π.χ. με βάση κάποια συγκεκριμένη πολιτική)
  - **thread-id σε κάθε BTB entry** για την αποφυγή πρόβλεψης branches που ανήκουν σε άλλα threads
  - **ξεχωριστή RAS για κάθε thread** για την πρόβλεψη της διεύθυνσης επιστροφής μετά από κλήση υπορουτίνας σε κάθε thread
  - **ξεχωριστός ROB για κάθε thread** προκειμένου το commit και η διαχείριση των mispredicted branches + των exceptions να γίνεται ανεξάρτητα για κάθε thread
  - **μεγαλύτερο register file**, για να υποστηρίζει λογικούς καταχωρητές για όλα τα threads + επιπλέον καταχωρητές για register renaming
  - **ξεχωριστό renaming table για κάθε thread**
    - *εφόσον οι λογικοί καταχωρητές για όλα τα threads απεικονίζονται όλοι σε διαφορετικούς φυσικούς καταχωρητές, οι εντολές από διαφορετικά threads μπορούν να αναμιχθούν μετά το renaming χωρίς να συγχέονται οι source και target operands ανάμεσα στα threads*

Case Studies

1. Pentium 4

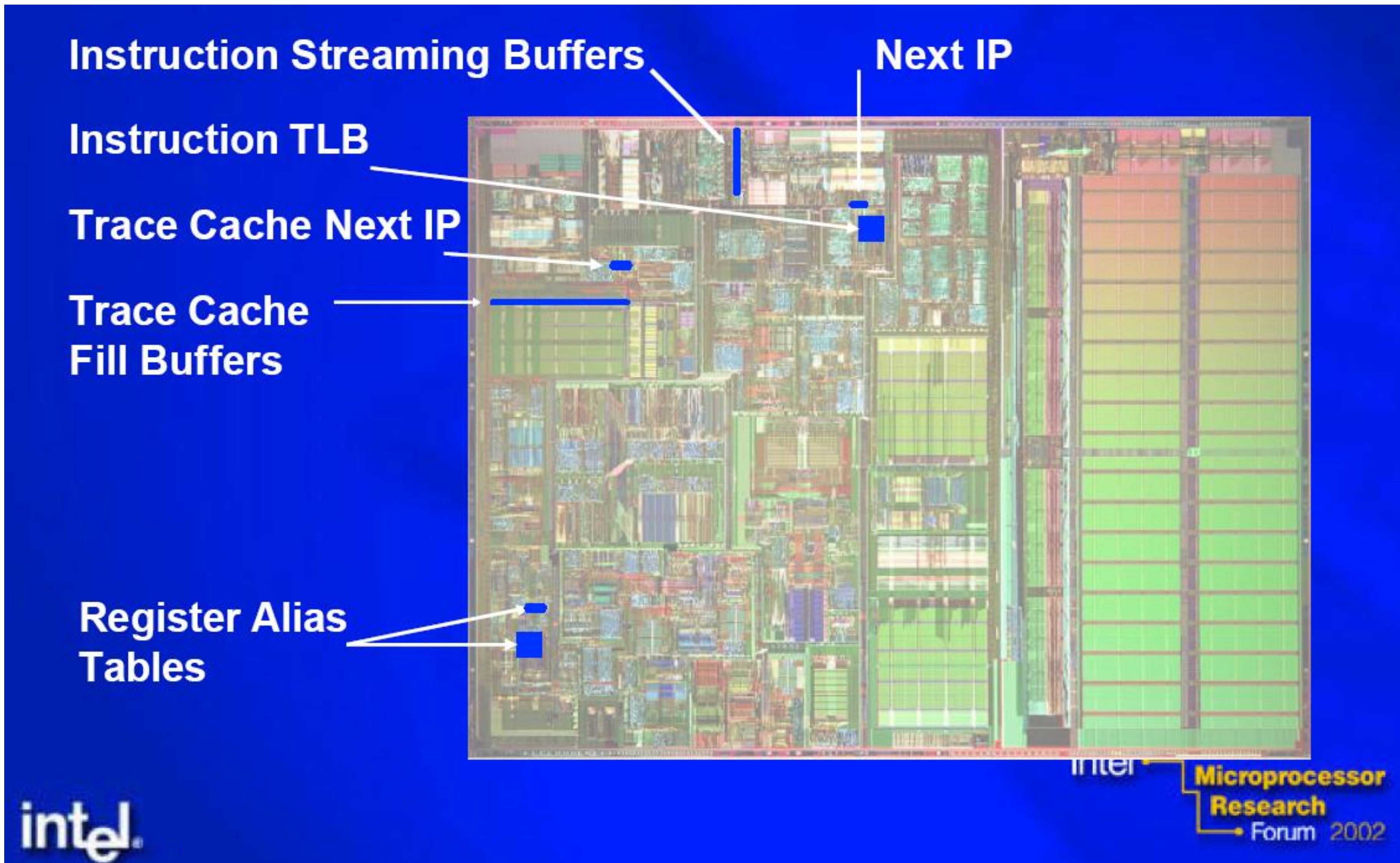
2. Power5

3. UltraSPARC T1

Case-study 1: Pentium 4
SMT in Hyperthreading technology (~2002)

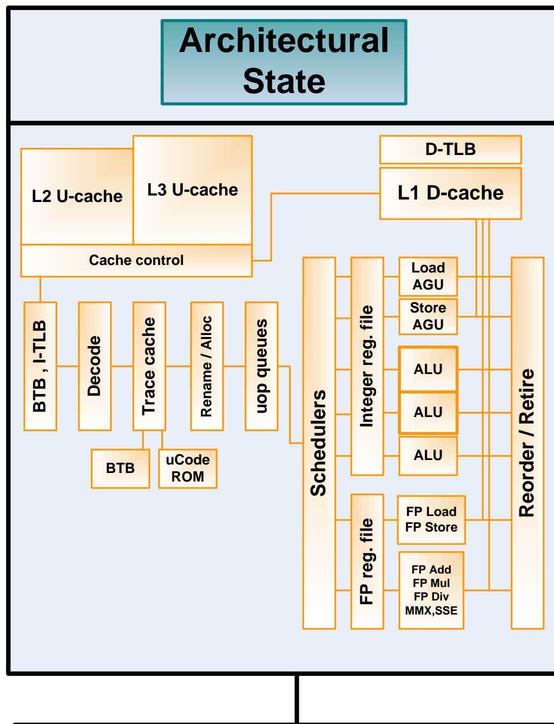
- 2-way SMT
  - το λειτουργικό «βλέπει» 2 CPUs
  - εκτελεί δύο διεργασίες ταυτόχρονα
    - » πολυπρογραμματιζόμενα φορτία
    - » πολυνηματικές εφαρμογές
- Ο φυσικός επεξεργαστής διατηρεί την αρχιτεκτονική για 2 λογικούς επεξεργαστές
- Επιπλέον κόστος για υποστήριξη 2 ταυτόχρονων νημάτων εκτέλεσης < 5% του αρχικού hardware

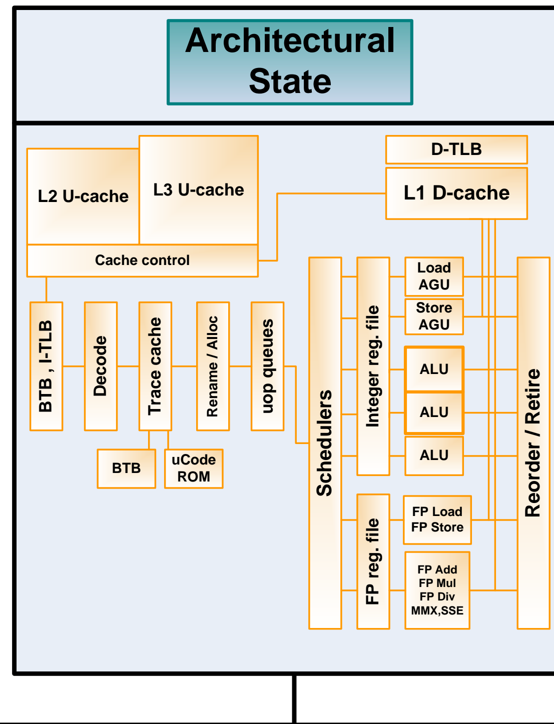
Τι προστέθηκε…



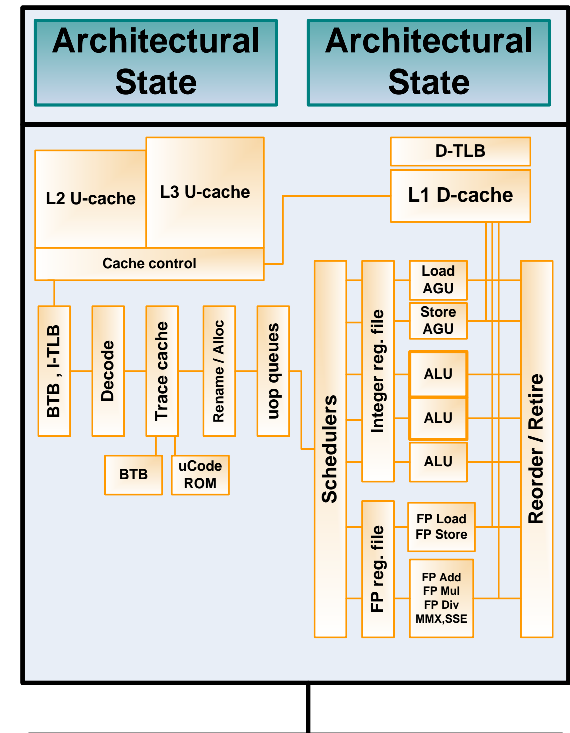
Επεξεργαστικοί πόροι: πολλαπλά αντίγραφα vs. διαμοιρασμός

Multiprocessor





Hyperthreading

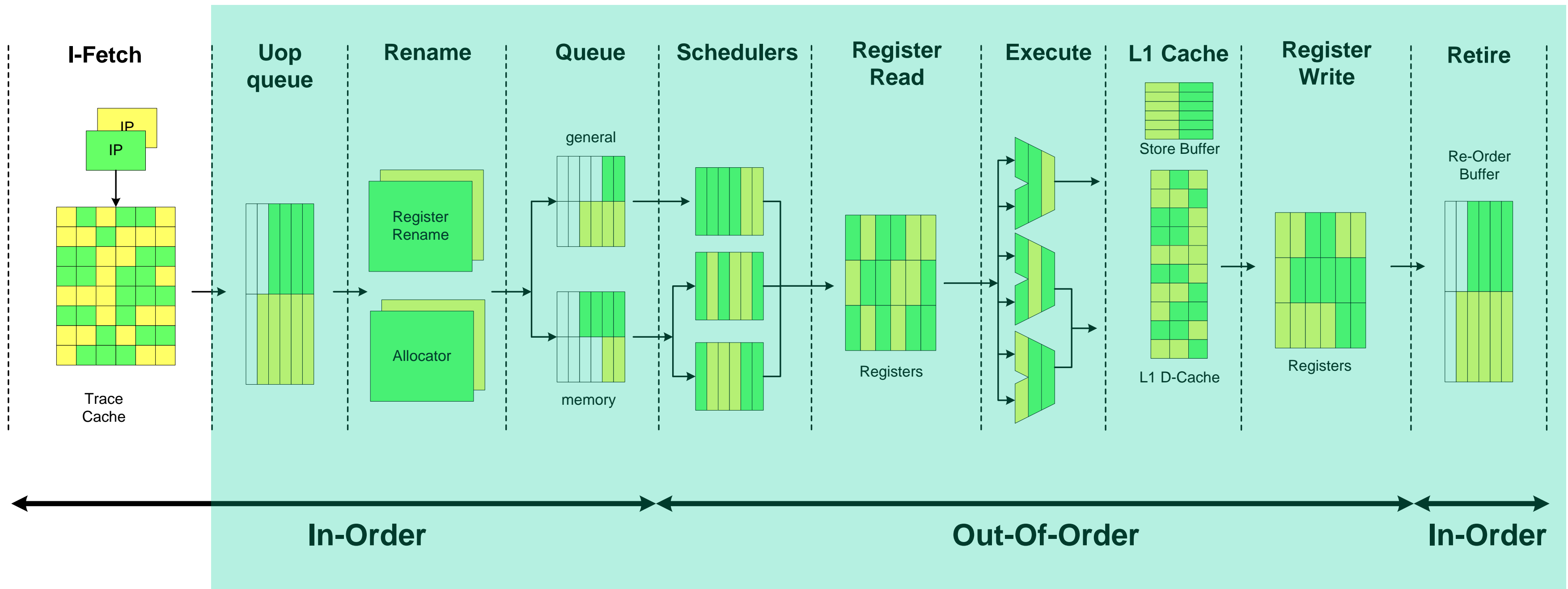


- στους πολυεπεξεργαστές τα resources βρίσκονται σε πολλαπλά αντίγραφα
- στο Hyper-threading τα resources διαμοιράζονται

Διαχείριση επεξεργαστικών πόρων

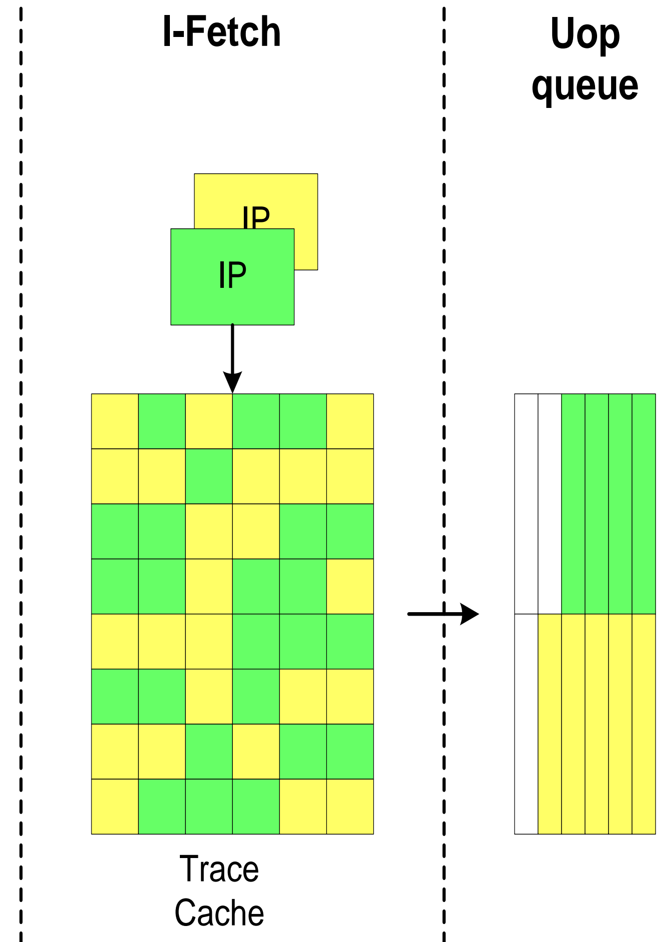
- Πόροι σε πολλαπλά αντίγραφα:
  - αρχιτεκτονική κατάσταση: GPRs, control registers, APICs
  - instruction pointers, renaming hardware
  - smaller resources: ITLBs, branch target buffer, return address stack
- Στατικά διαχωρισμένοι πόροι:
  - ROB, Load/Store queues, instruction queues
  - κάθε νήμα μπορεί να χρησιμοποιήσει έως τα μισά (το πολύ) entries κάθε τέτοιου πόρου
    - » ένα νήμα δεν μπορεί να οικειοποιηθεί το σύνολο των entries, στερώντας τη δυνατότητα από το άλλο νήμα να συνεχίσει την εκτέλεσή του
    - » εξασφαλίζεται η απρόσκοπτη πρόοδος ενός νήματος, ανεξάρτητα από την πρόοδο του άλλου νήματος
- Δυναμικά διαμοιραζόμενοι πόροι (κατ' απαίτηση):
  - out-of-order execution engine, caches

Pentium 4 w/ Hyper-Threading: Front End

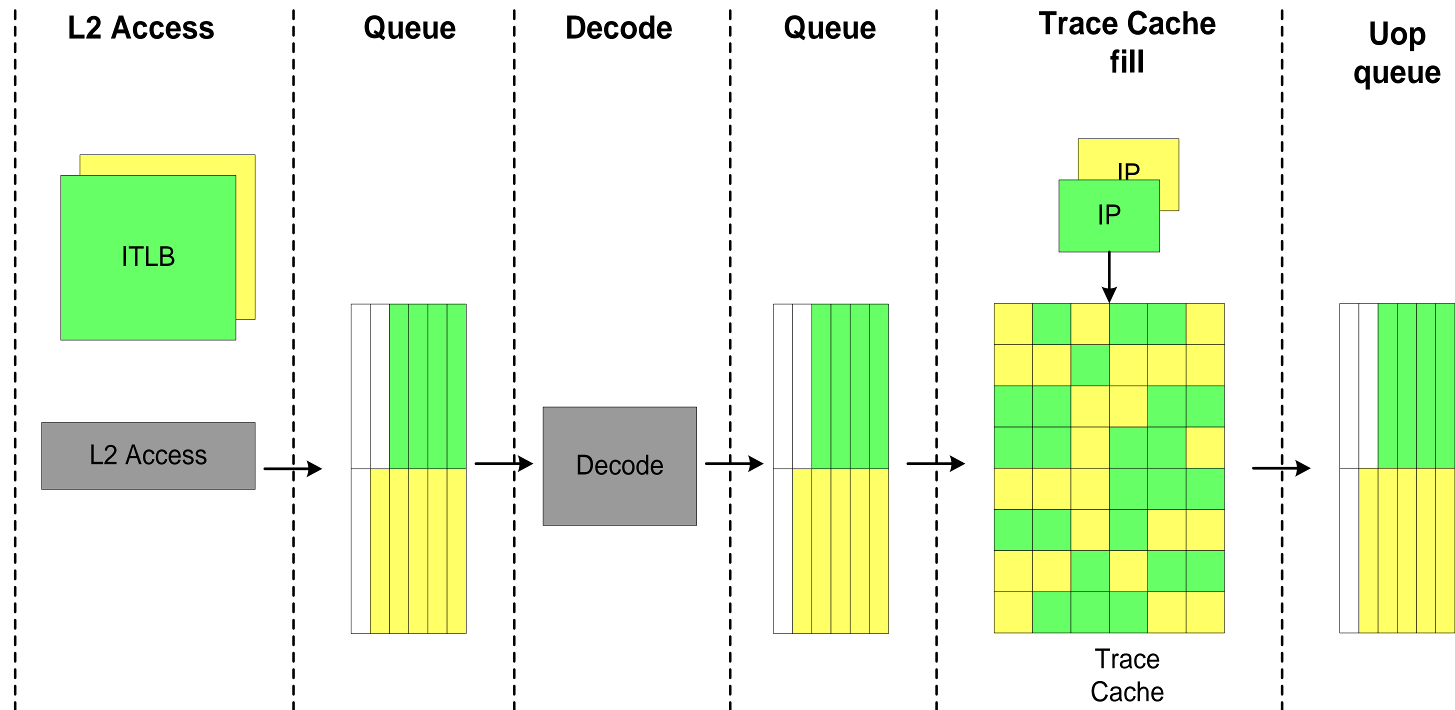


Pentium 4 w/ Hyper-Threading: Front End – trace cache hit

- Trace cache
  - «ειδική» instruction cache που κρατάει αποκωδικοποιημένες μικρο-εντολές
  - σε κάθε cache line αποθηκεύονται οι μικρο-εντολές στη σειρά με την οποία εκτελούνται (π.χ. εντολή άλματος μαζί με την ακολουθία εντολών στην προβλεφθείσα κατεύθυνση)
- Σε ταυτόχρονη ζήτηση από τους 2 LPs (Logical Processors), η πρόσβαση εναλλάσσεται κύκλο-ανά-κύκλο
- Όταν μόνο 1 LP ζητάει πρόσβαση, μπορεί να χρησιμοποιήσει την TC στο μέγιστο δυνατό fetch bandwidth (3 μIPC)

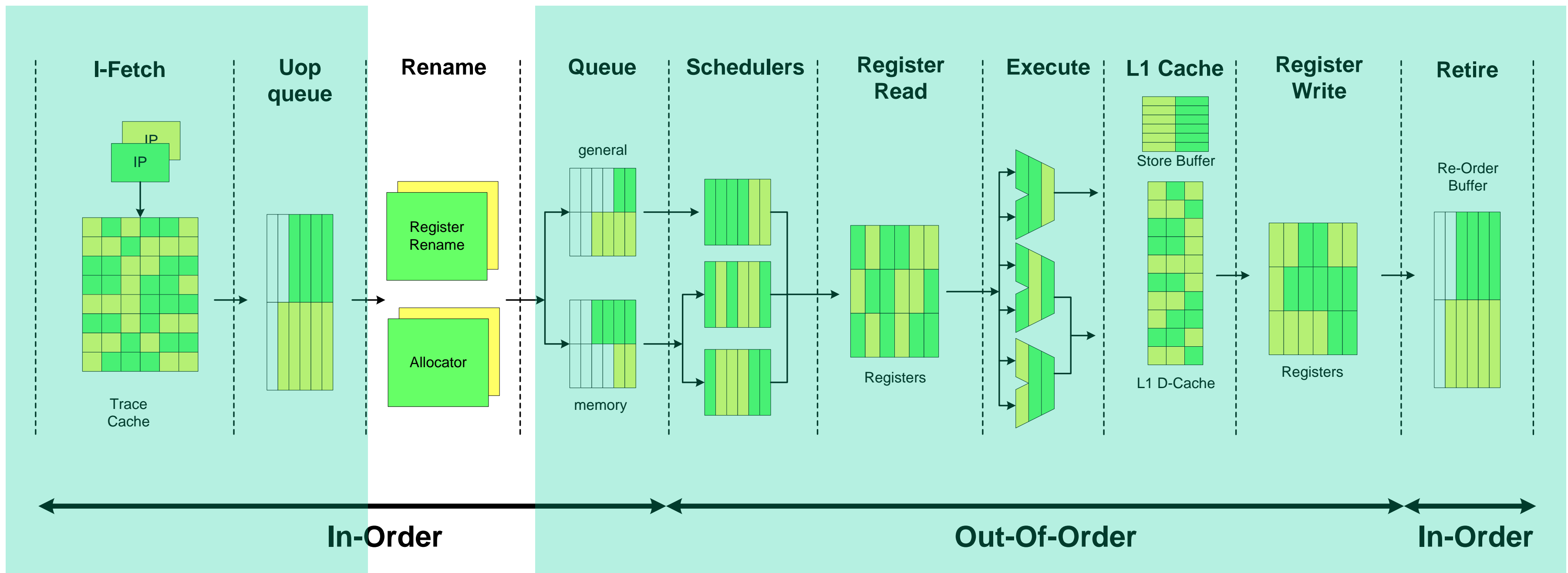


Pentium 4 w/ Hyper-Threading: Front End – trace cache miss



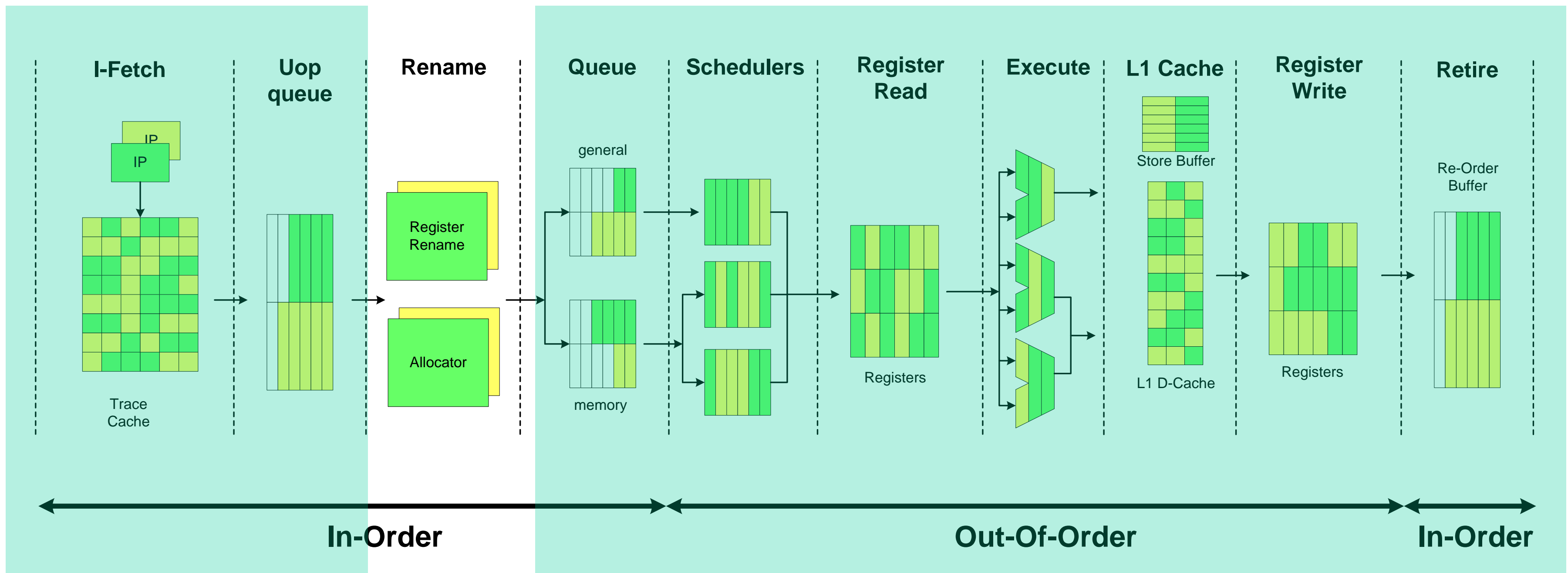
- L2 cache
  - η πρόσβαση γίνεται με FCFS τρόπο
- Decode
  - σε ταυτόχρονη ζήτηση από τους 2 LPs, η πρόσβαση εναλλάσσεται, αλλά με πιο coarse-grained τρόπο (δηλ. όχι κύκλο-ανά-κύκλο, αλλά k κύκλους-ανά-k κύκλους)

Pentium 4 w/ Hyper-Threading: Execution engine



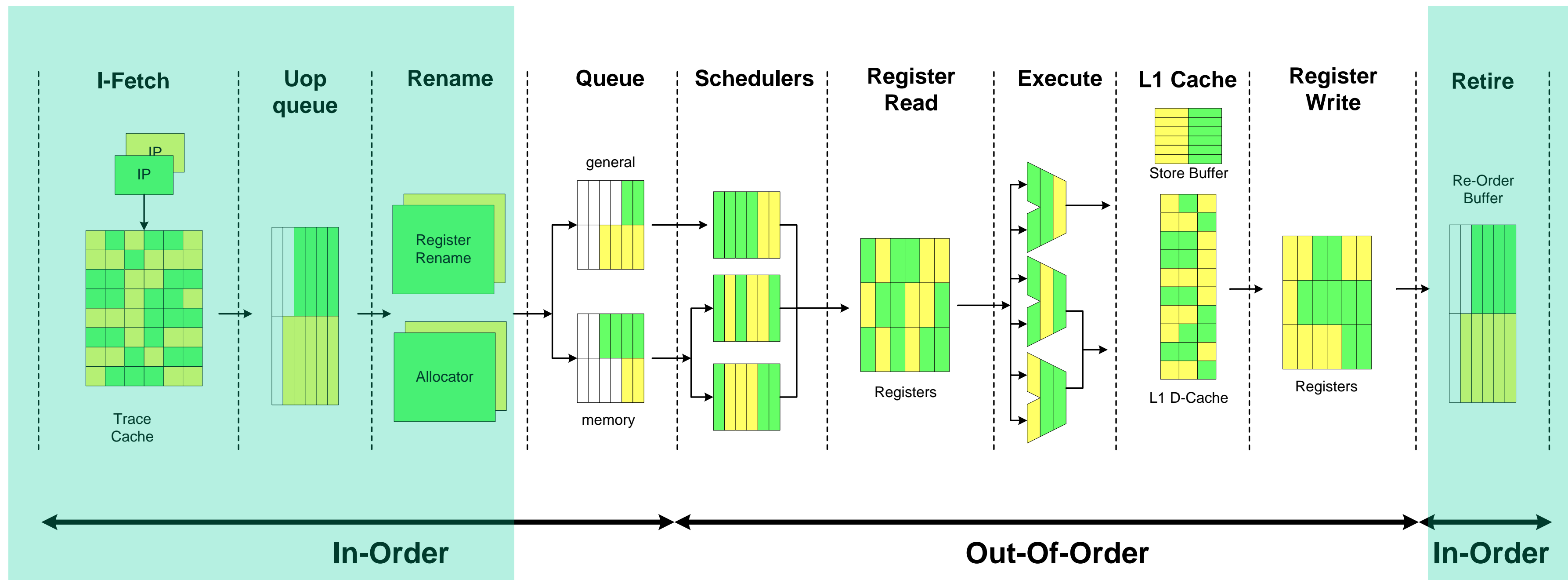
- Allocator: εκχωρεί entries σε κάθε LP
  - 63/126 ROB entries
  - 24/48 Load buffer entries
  - 12/24 Store buffer entries
  - 128/128 Integer physical registers
  - 128/128 FP physical registers
- Σε ταυτόχρονη ζήτηση από τους 2 LPs, η πρόσβαση εναλλάσσεται κύκλο-ανά-κύκλο
- stall-άρει έναν LP όταν επιχειρεί να χρησιμοποιήσει περισσότερα από τα μισά entries των στατικά διαχωρισμένων πόρων

Pentium 4 w/ Hyper-Threading: Execution engine



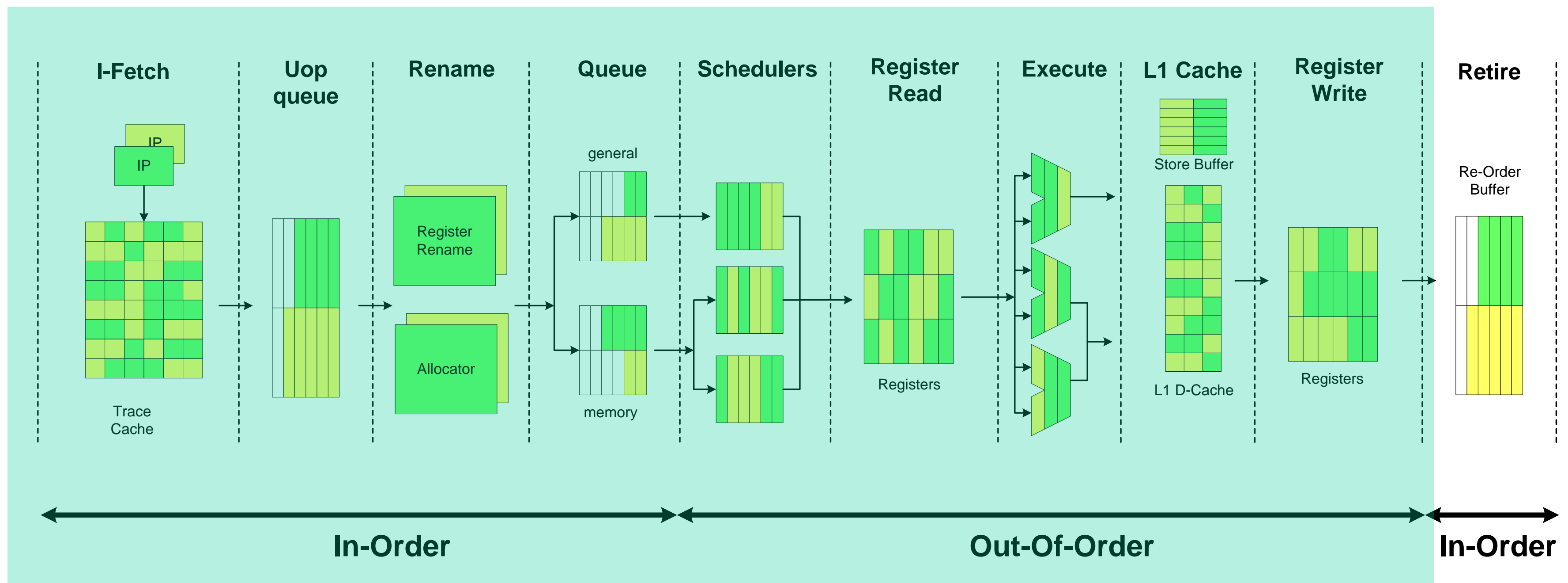
- Register renaming unit
  - επεκτείνει δυναμικά τους architectural registers απεικονίζοντάς τους σε ένα μεγαλύτερο σύνολο από physical registers
  - ξεχωριστό register map table για κάθε LP

Pentium 4 w/ Hyper-Threading: Execution engine



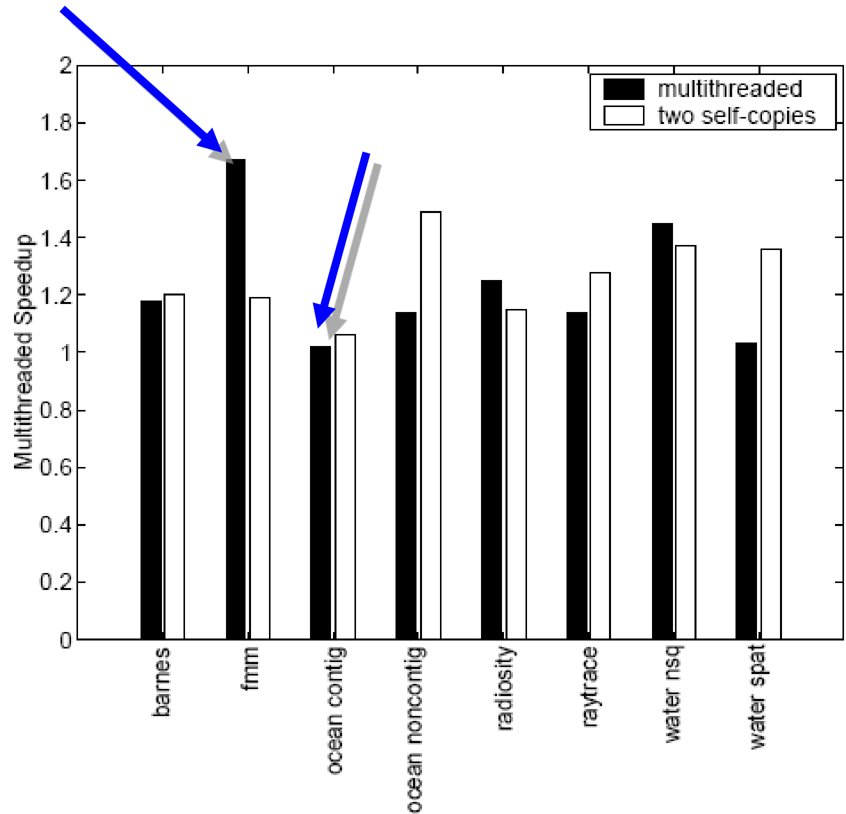
- Schedulers / Execution units
  - στην πραγματικότητα δε (χρειάζεται να) ξέρουν σε ποιον LP ανήκει η εντολή που εκτελούν
  - general+memory queues στέλνουν μικρο-εντολές στους δρομολογητές, με την πρόσβαση να εναλλάσσεται κύκλο-ανά-κύκλο ανάμεσα στους 2 LPs
  - 6 μIPC dispatch/execute bandwidth (→ 3 μIPC per-LP effective bandwidth, όταν και οι 2 LPs είναι ενεργοί)

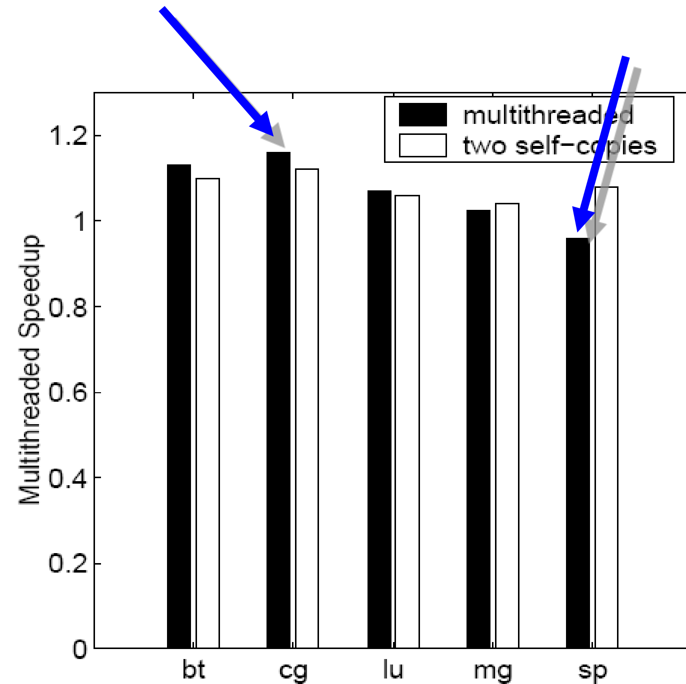
Pentium 4 w/ Hyper-Threading: Retirement



- Η αρχιτεκτονική κατάσταση κάθε LP γίνεται commit με τη σειρά προγράμματος, εναλλάσσοντας την πρόσβαση στον ROB ανάμεσα στους 2 LPs κύκλο-ανά-κύκλο
- 3 μIPC retirement bandwidth

Multithreaded speedup

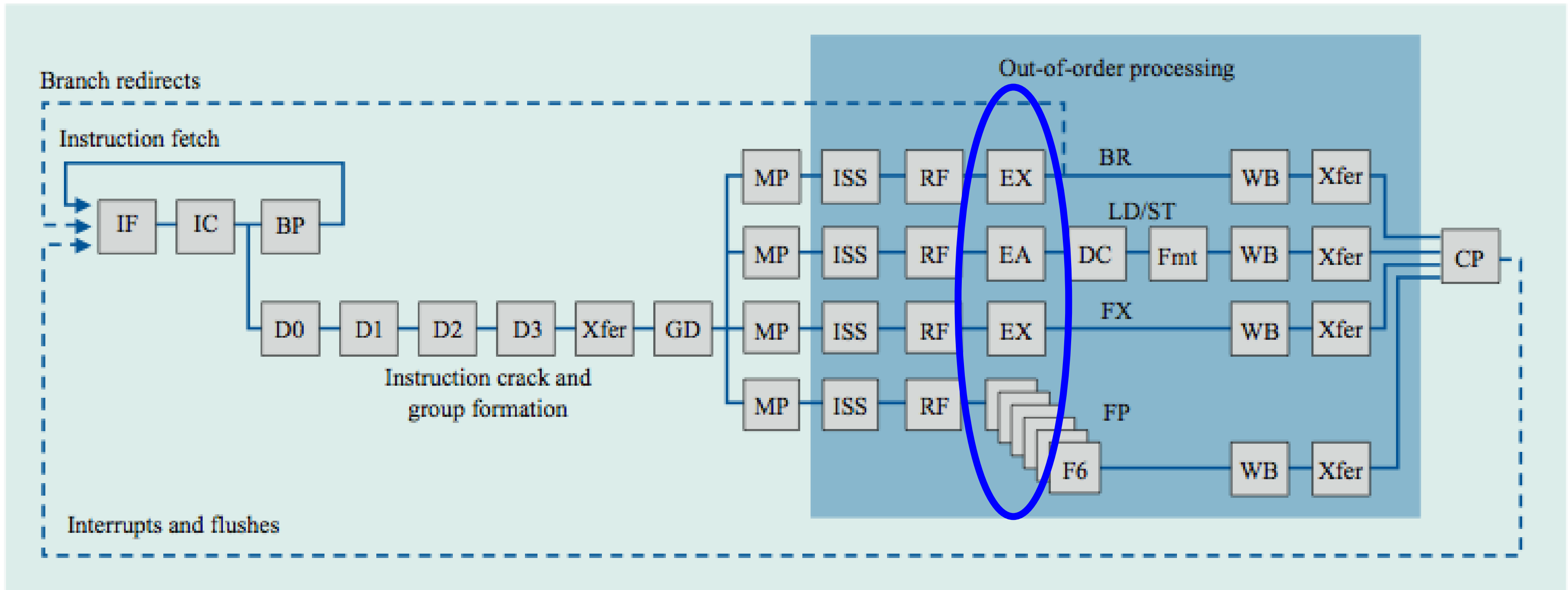




From: Tuck and Tullsen, "Initial Observations of the Simultaneous Multithreading Pentium 4 Processor", PACT 2003.

- SPLASH2 Benchmarks: 1.02 – 1.67
- NAS Parallel Benchmarks: 0.96 – 1.16

Case-study 2: Power5
Επέκταση του Power4 για υποστήριξη SMT (2004)

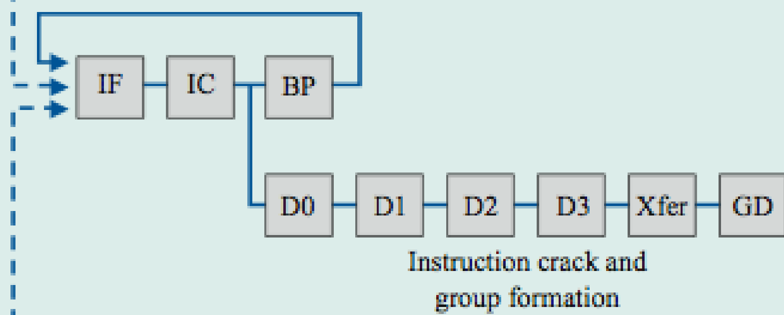


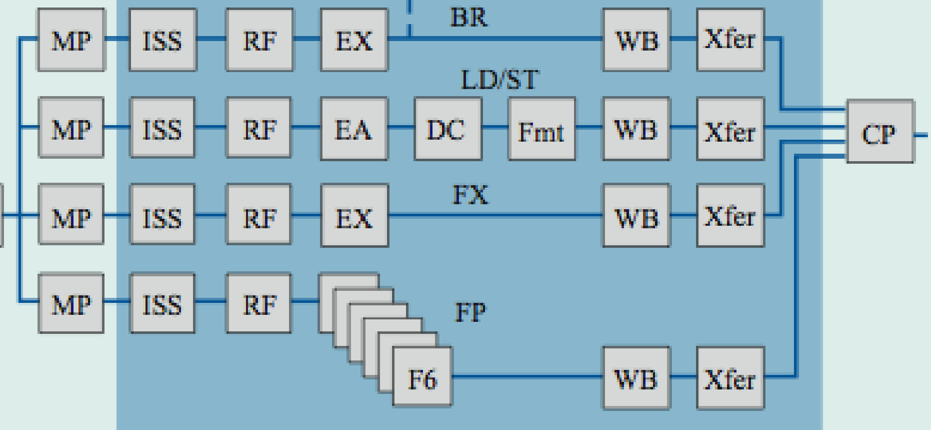
- Power4: Single-threaded «προκάτοχος» του Power5
- 8 execution units
  - 2 Float. Point, 2 Load/Store, 2 Fixed Point, 1 Branch, 1 Conditional Reg. unit
  - κάθε μία μπορεί να κάνει issue 1 εντολή ανά κύκλο
- Execution bandwidth: 8 operations ανά κύκλο
  - (1 fpadd + 1 fpmult) x 2FP + 1 load/store x 2 LD/ST + 1 integer x 2 FX

Power4

Branch redirects

Out-of-order processing



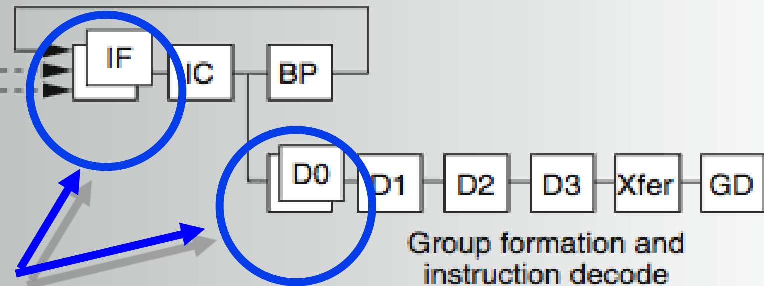


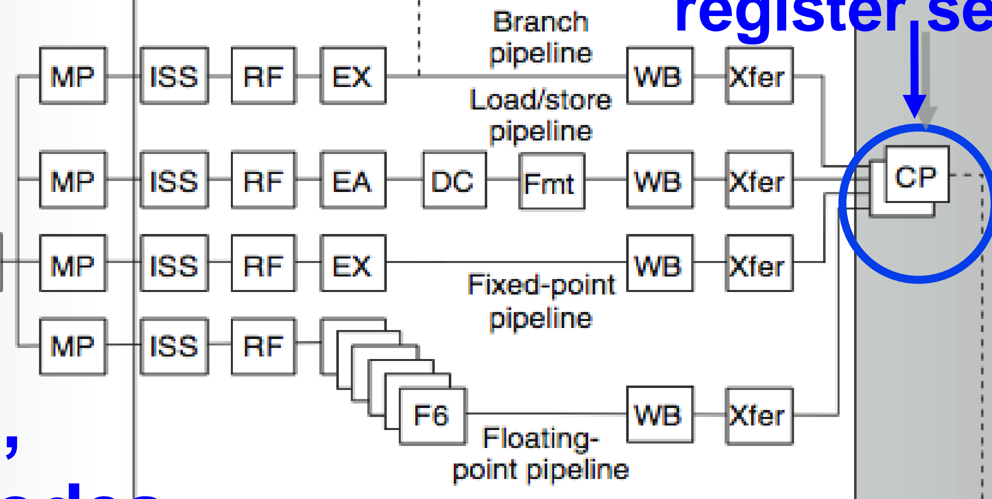
Power5

Branch redirects

Out-of-order processing

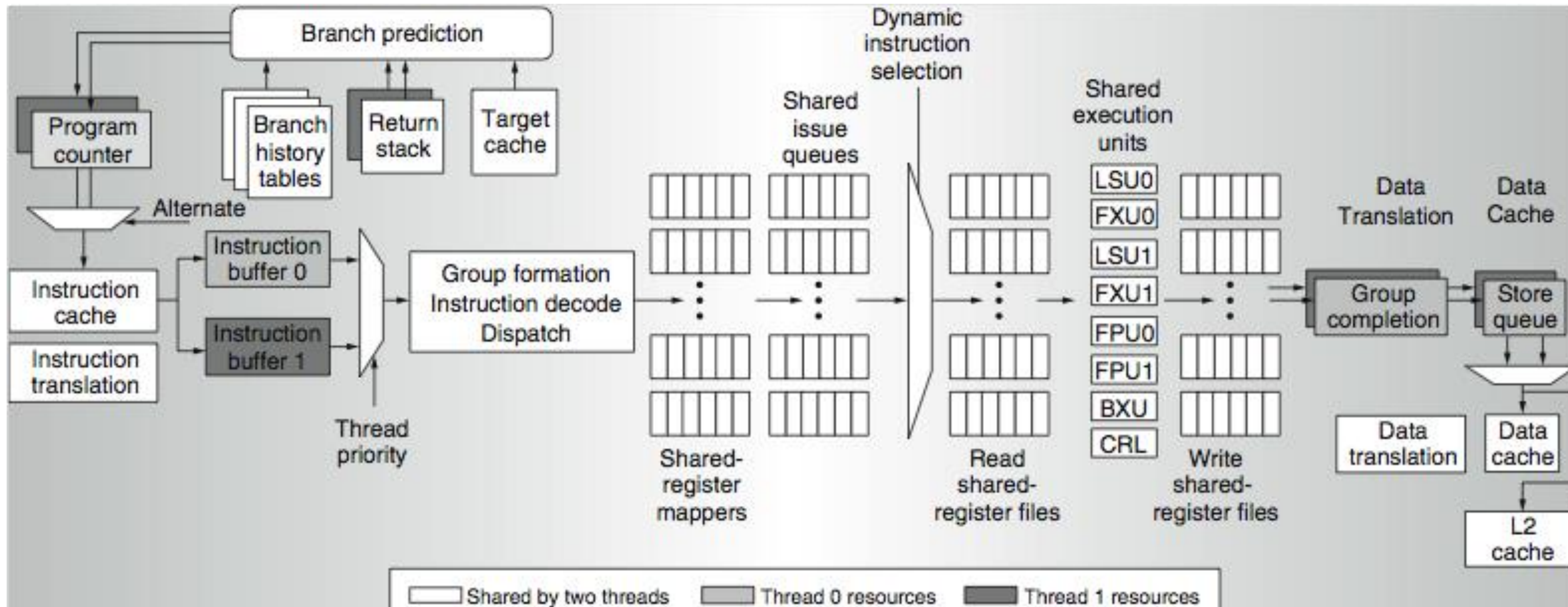
2 commits (architected register sets)





2 fetch (PC),
2 initial decodes

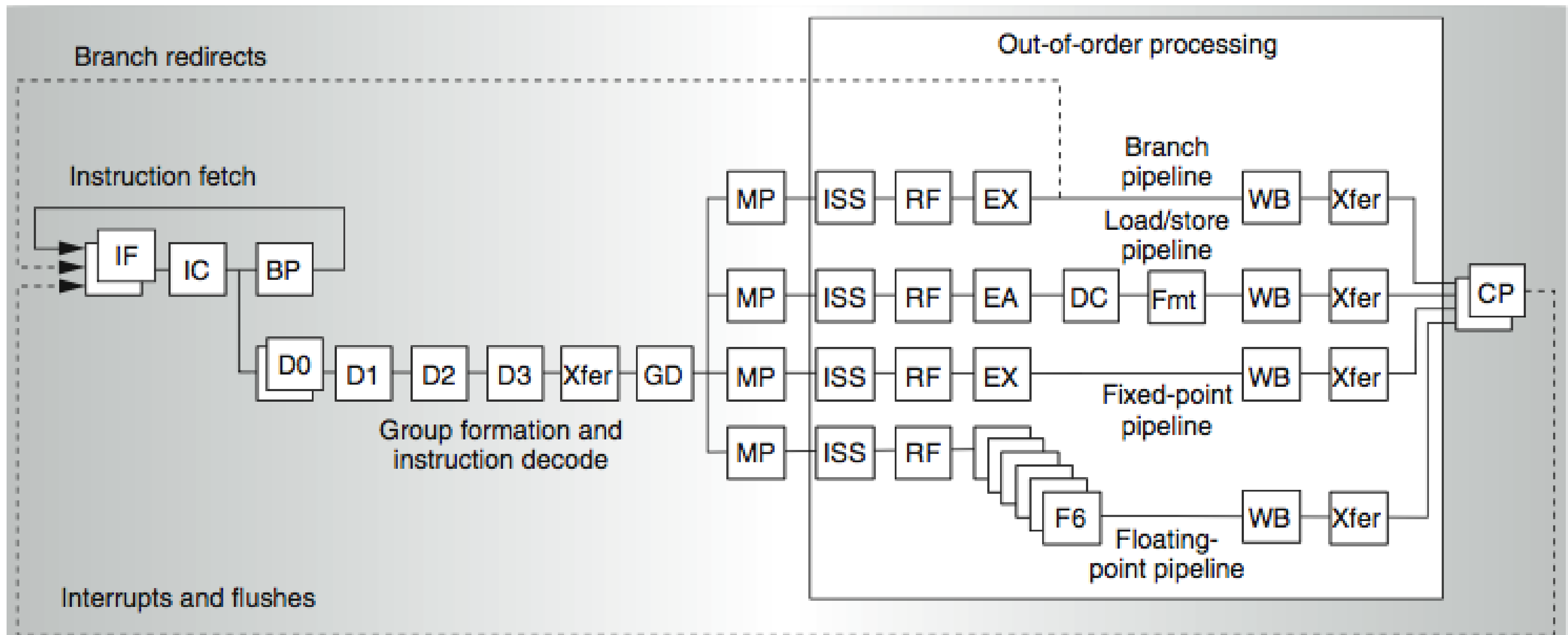
SMT resource management



Αλλαγές στον Power5 για να υποστηρίζεται το SMT

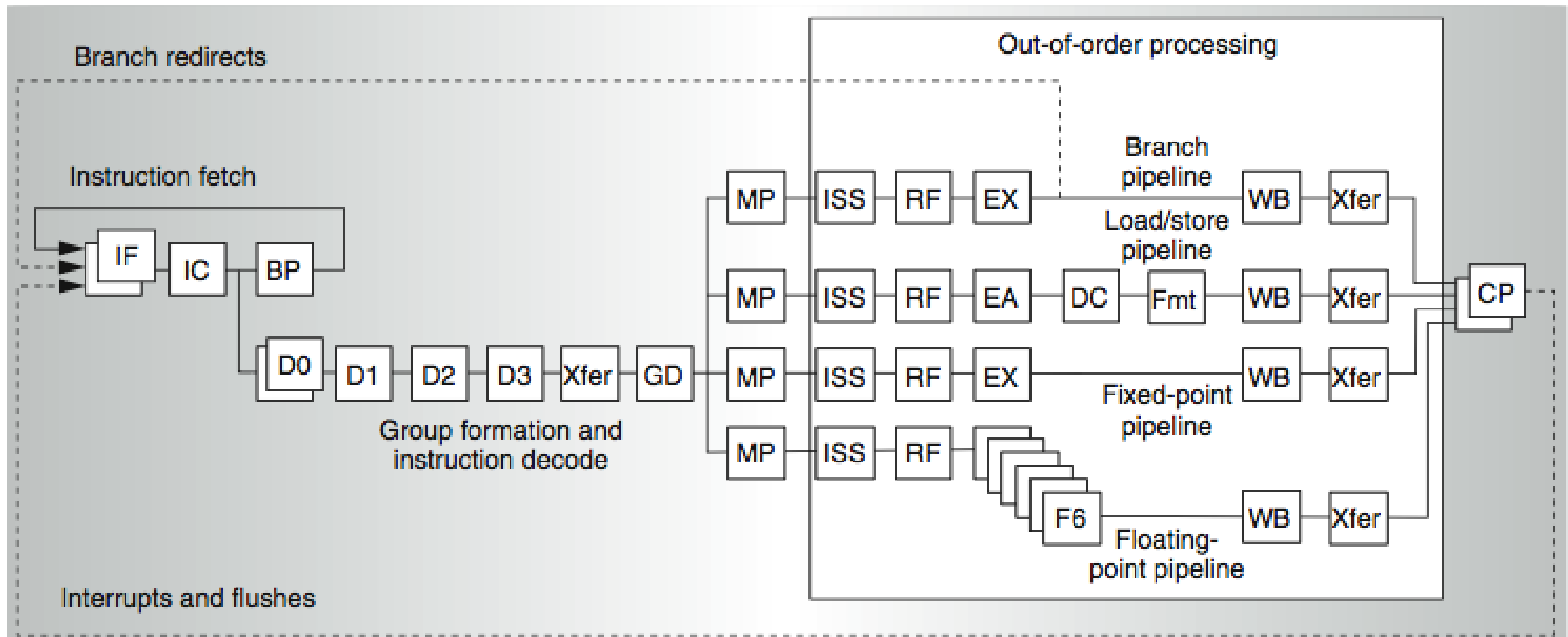
- Αύξηση συσχετιστικότητας της L1 instruction cache και των ITLBs
- Ξεχωριστές Load/Store queues για κάθε νήμα
- Αύξηση μεγέθους των L2 (1.92 vs. 1.44 MB) και L3 caches
- Ξεχωριστό instruction prefetch hardware και instruction buffers για κάθε νήμα
- Αύξηση των registers από 152 σε 240
- Αύξηση του μεγέθους των issue queues
- Αύξηση μεγέθους κατά 24% σε σχέση με τον Power4 εξαιτίας της προσθήκης hardware για υποστήριξη SMT

Power 5 datapath



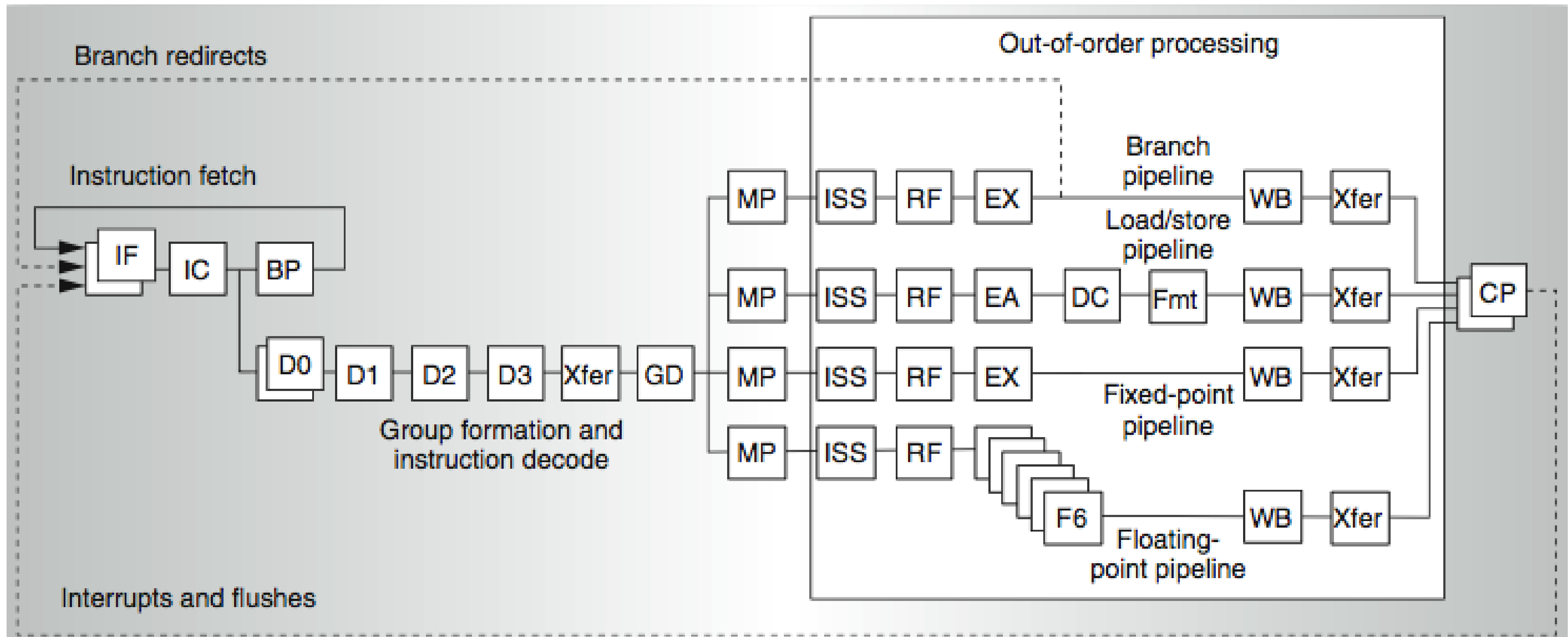
- Στο IF στάδιο η πρόσβαση εναλλάσσεται κύκλο-ανά-κύκλο ανάμεσα στα 2 threads
  - 2 instruction fetch address registers, 1 για κάθε νήμα
- Μπορούν να φορτωθούν 8 instructions σε κάθε κύκλο (στάδιο IC) από την I-Cache
  - σε έναν συγκεκριμένο κύκλο, οι εντολές που φορτώνονται προέρχονται όλες από το ίδιο thread
  - και τοποθετούνται στο instruction buffer του thread αυτού (στάδιο D0)

Power 5 datapath



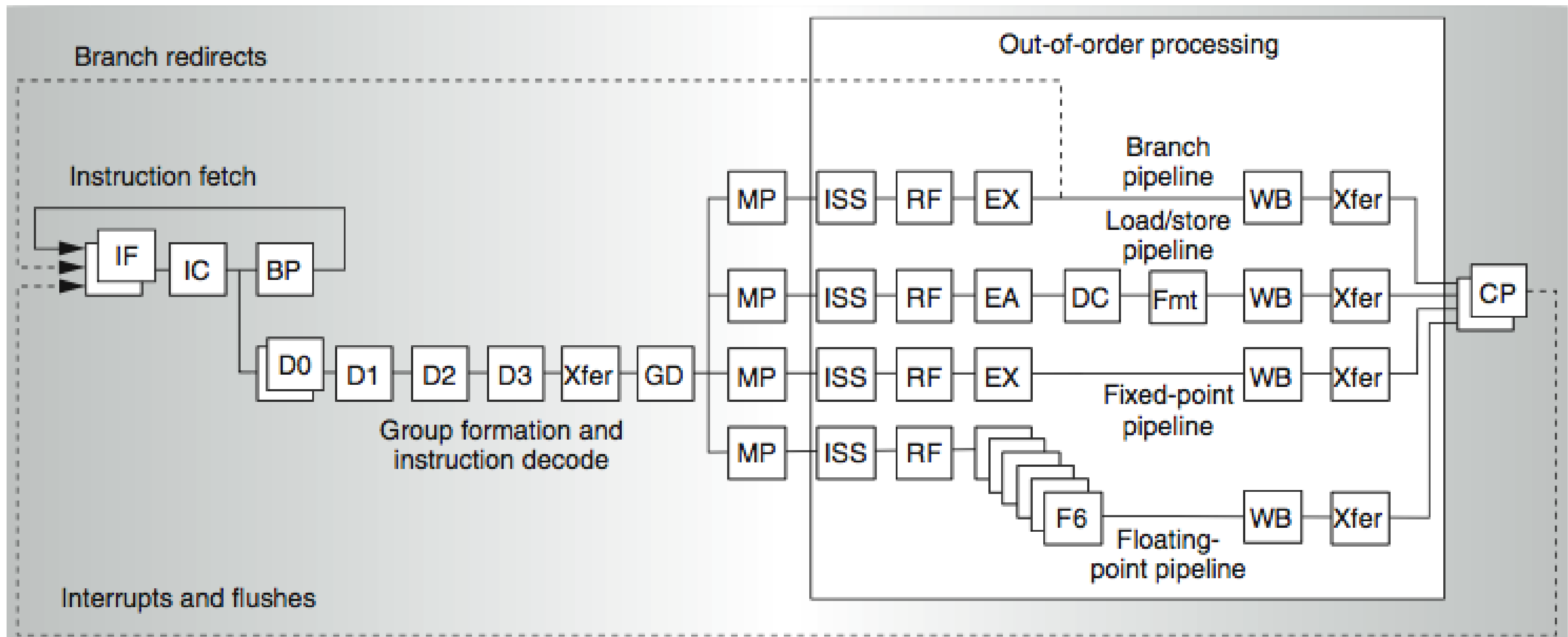
- Στα στάδια D1-D3, ανάλογα με την προτεραιότητα κάθε thread, ο επεξεργαστής διαλέγει εντολές από έναν από τους δύο instruction buffers και σχηματίζει ένα group
- Όλες οι εντολές σε ένα group προέρχονται από το ίδιο thread και αποκωδικοποιούνται παράλληλα
- Κάθε group μπορεί να περιέχει το πολύ 5 εντολές

Power 5 datapath



- Όταν όλοι οι απαιτούμενοι πόροι γίνονται διαθέσιμοι για τις εντολές ενός group, τότε το group μπορεί να γίνει dispatch (στάδιο GD)
  - το group τοποθετείται στο Global Completion Table (ROB)
  - τα entries στο GCT εκχωρούνται με τη σειρά προγράμματος για κάθε thread, και απελευθερώνονται (πάλι με τη σειρά προγράμματος) μόλις το group γίνει commit
- Μετά το dispatch, κάθε εντολή του group διέρχεται μέσα από το register renaming στάδιο (MP)
  - 120 physical GPRs, 120 physical FPRs
  - τα 2 νήματα διαμοιράζονται δυναμικά registers

Power 5 datapath



- Issue, execute, write-back
  - δε γίνεται διάκριση ανάμεσα στα 2 threads
- Group completion (στάδιο CP)
  - 1 group commit ανά κύκλο για κάθε thread
  - στη σειρά προγράμματος του κάθε thread

Δυναμική εξισορρόπηση επεξεργαστικών πόρων

- Πρέπει να εξασφαλιστεί η απρόσκοπτη και ομαλή ροή των threads στο pipeline, ανεξάρτητα από τις απαιτήσεις του καθενός σε επεξεργαστικούς πόρους

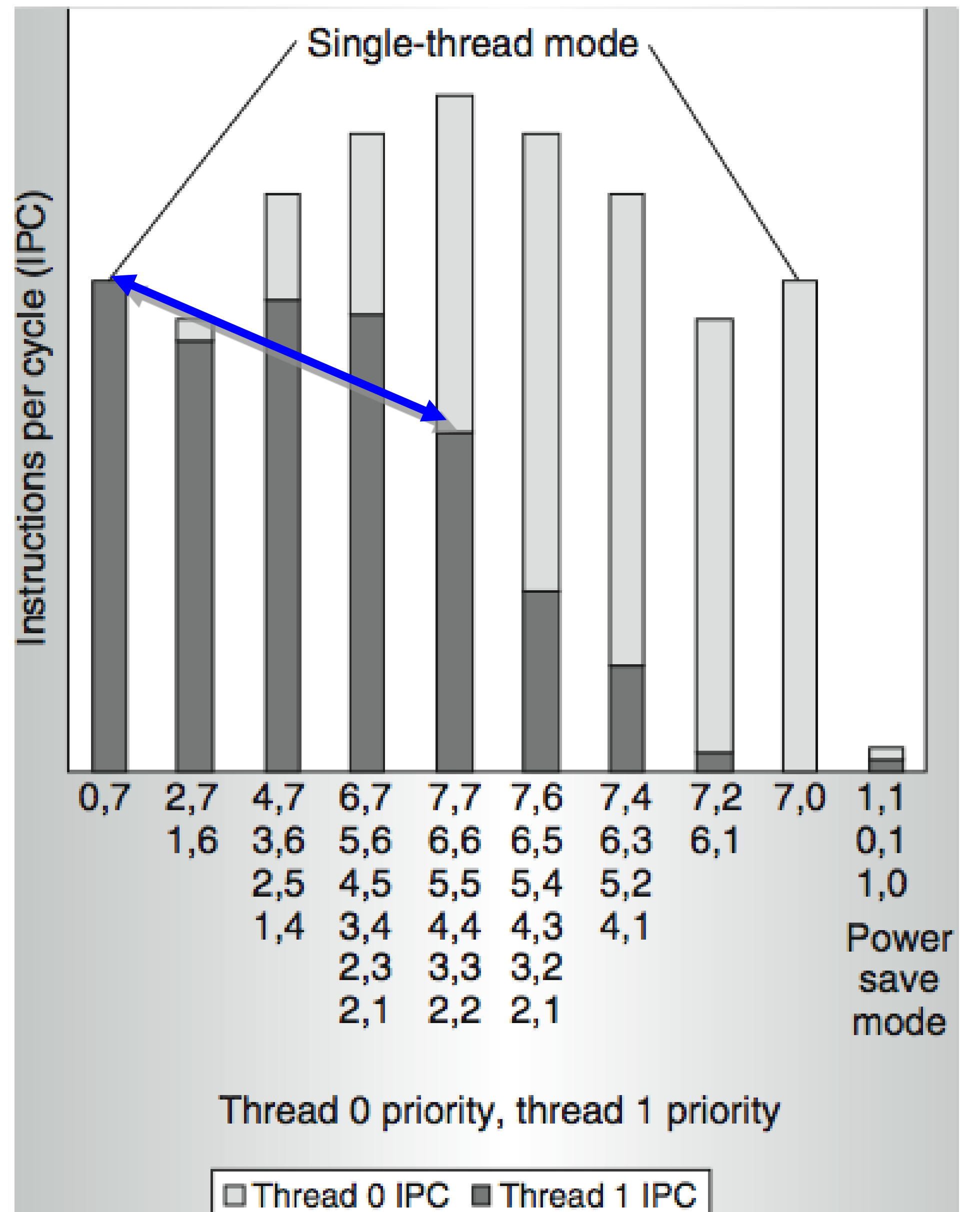
- Μηχανισμοί περιορισμού ενός πολύ απαιτητικού thread:
  - *μείωση προτεραιότητας του thread*, όταν διαπιστώνεται ότι τα GCT entries που χρησιμοποιεί ξεπερνούν ένα καθορισμένο όριο
  - όταν ένα thread έχει πολλά L2 misses, τότε οι μεταγενέστερες εξαρτώμενες εντολές του μπορούν να γεμίσουν τις issue queues, εμποδίζοντας να γίνουν dispatch εντολές από το άλλο thread
    - » *παρακολούθηση της Load Miss Queue* ενός thread έτσι ώστε όταν τα misses του υπερβαίνουν κάποιο όριο, *η αποκωδικοποίηση εντολών του να σταματάει* μέχρι να αποσυμφορηθούν οι issue queues
  - συμφόρηση στις issue queues μπορεί να συμβεί και όταν ένα thread εκτελεί μια εντολή που απαιτεί πολύ χρόνο
    - » *flushing των εντολών του thread* που περιμένουν να γίνουν dispatch και *προσωρινή διακοπή της αποκωδικοποίησης* εντολών του μέχρι να αποσυμφορηθούν οι issue queues

Ρυθμιζόμενη προτεραιότητα threads

- Επιτρέπει στο software να καθορίσει πότε ένα thread θα πρέπει να έχει μεγαλύτερο ή μικρότερο μερίδιο επεξεργαστικών πόρων
- Πιθανές αιτίες για διαφορετικές προτεραιότητες:
  - ένα thread εκτελεί ένα spin-loop περιμένοντας να πάρει κάποιο lock → δεν κάνει χρήσιμη δουλειά όσο spin-άρει
  - ένα thread δεν έχει δουλειά να εκτελέσει και περιμένει να του ανατεθεί δουλειά σε ένα idle loop
  - μία εφαρμογή πρέπει να τρέχει πιο γρήγορα σε σχέση με μία άλλη (π.χ. real-time application vs. background application)
- 8 software-controlled επίπεδα προτεραιότητας για κάθε thread
- Ο επεξεργαστής παρατηρεί τη διαφορά των επιπέδων προτεραιότητας των threads, και δίνει στο thread με τη μεγαλύτερη προτεραιότητα περισσότερους διαδοχικούς κύκλους για αποκωδικοποίηση εντολών του

Επίδραση προτεραιότητας στην απόδοση κάθε thread

- Όταν τα threads έχουν ίδιες προτεραιτότητες, εκτελούνται πιο αργά απ'ό,τι αν κάθε thread είχε διαθέσιμα όλα τα resources του επεξεργαστή (single-thread mode)



"Large" vs. "Small" Cores

Large Core

- Out-of-order
- Wide fetch e.g. 4-wide
- Deeper pipeline
- Aggressive branch predictor (e.g. hybrid)
- Multiple functional units
- Trace cache
- Memory dependence speculation

Small Core

- In-order
- Narrow Fetch e.g. 2-wide
- Shallow pipeline
- Simple branch predictor (e.g. Gshare)
- Few functional units

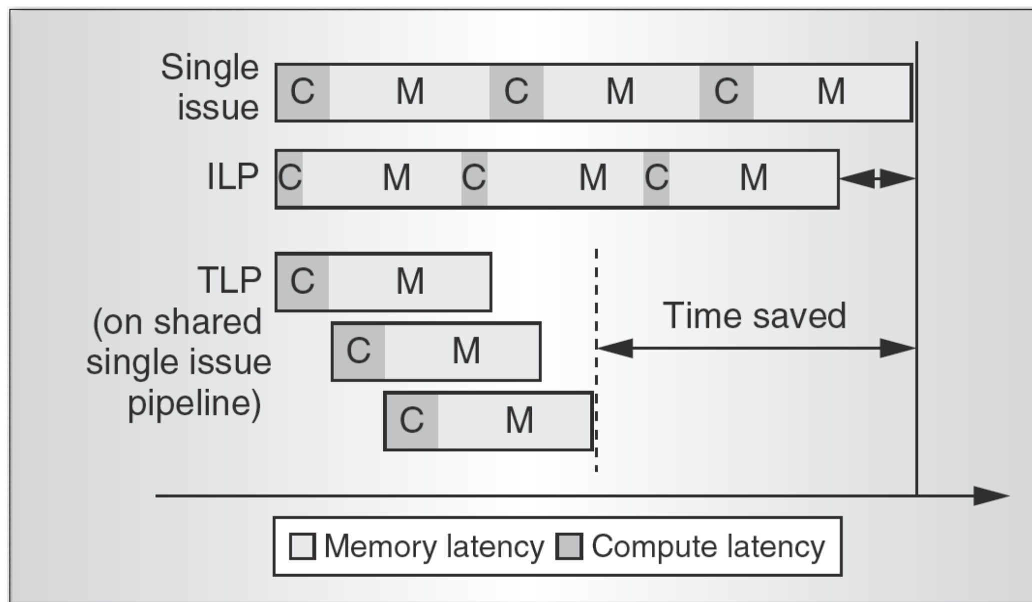
Large Cores are power inefficient:
e.g., 2x performance for 4x area (power)

Large vs. Small Cores

- Grochowski et al., "*Best of both Latency and Throughput*," ICCD 2004.

	Large core	Small core
Microarchitecture	Out-of-order, 128-256 entry ROB	In-order
Width	3-4	1
Pipeline depth	20-30	5
Normalized performance	5-8x	1x
Normalized power	20-50x	1x
Normalized energy/instruction	4-6x	1x

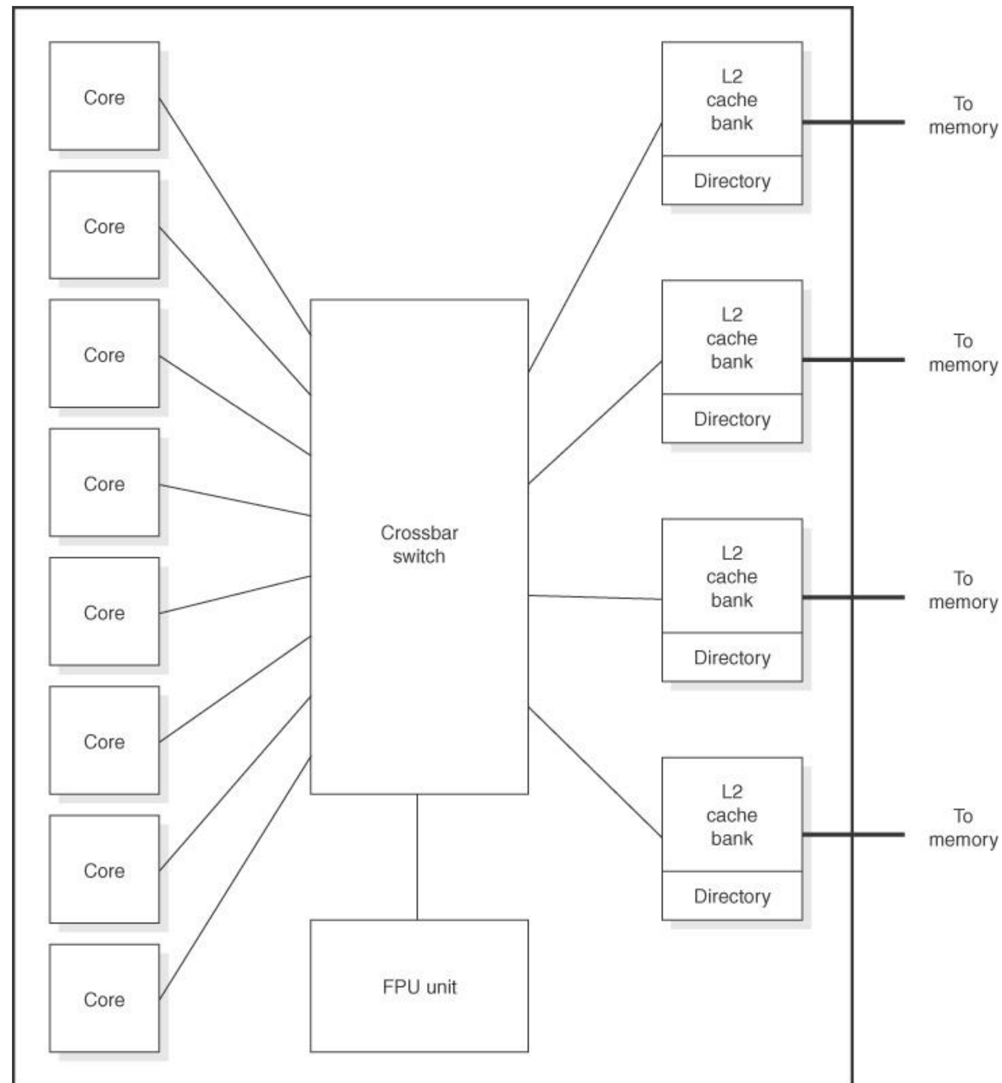
Case-study 3: UltraSPARC T1 (“Niagara”) (2005)



- Συμπεριφορά επεξεργαστών βελτιστοποιημένων για TLP και ILP σε server workloads:
  - server workloads:
    - » υψηλός TLP (μεγάλος αριθμός παράλληλων client requests)
    - » χαμηλός ILP (υψηλά miss rates, πολλά unpredictable branches, συχνές load-load εξαρτήσεις)
    - » το memory access time κυριαρχεί στο συνολικό χρόνο εκτέλεσης
- Ο “ILP” επεξεργαστής μειώνει μόνο το computation time
  - το memory access time κυριαρχεί σε ακόμα μεγαλύτερο ποσοστό
- Στον “TLP” επεξεργαστή, το memory access ενός thread επικαλύπτεται από computations από άλλα threads
  - η απόδοση αυξάνεται για μια memory-bound multithreaded εφαρμογή

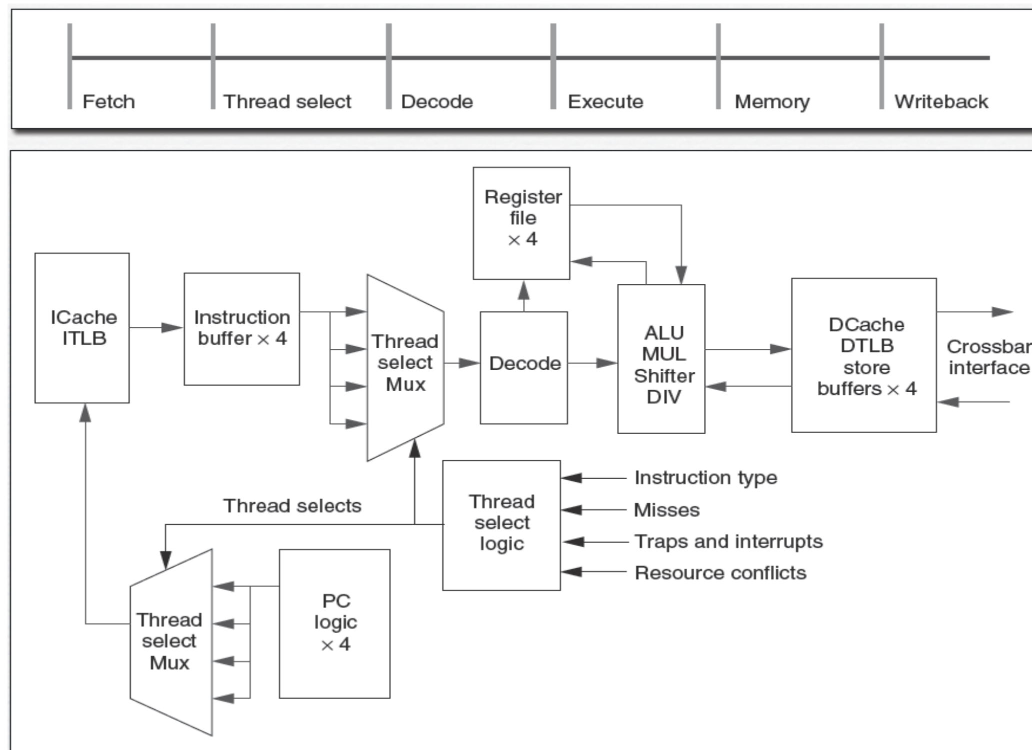
UltraSPARC T1

- in-order, single-issue
  - επικεντρώνεται πλήρως στην εκμετάλλευση του TLP
- 4-8 cores, 4 threads ανά core
  - max 32 threads
  - fine-grained multithreading
- L1D + L1I μοιραζόμενες από τα 4 threads
- L2 cache + FPU μοιραζόμενη από όλα τα threads
- ξεχωριστό register set + instruction buffers + store buffers για κάθε thread



© 2007 Elsevier, Inc. All rights reserved.

UltraSPARC T1 pipeline

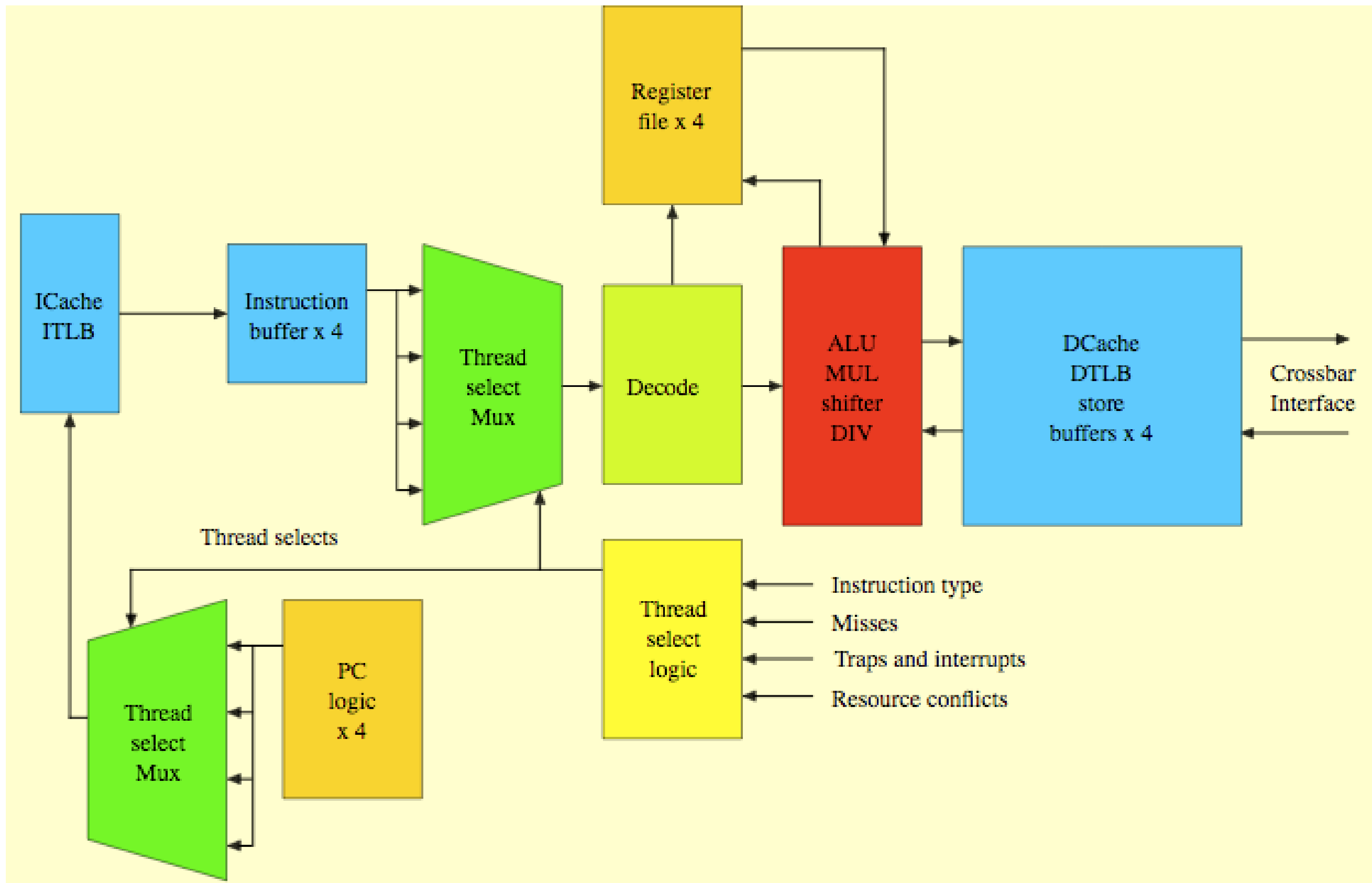


- Όπως το κλασικό 5-stage pipeline + thread select stage
- Fetch stage: ο thread select mux επιλέγει ποιος από τους 4 PCs θα πρέπει να προσπελάσει την ICache και το ITLB
- Thread select stage: αποφασίζει σε κάθε κύκλο ποιος από τους 4 instruction buffers θα τροφοδοτήσει με εντολές τα επόμενα στάδια
  - αν το thread-select στάδιο επιλέξει ένα νήμα από το οποίο θα στείλει εντολές, το fetch στάδιο θα επιλέξει το ίδιο thread για να προσπελάσει την ICache

UltraSPARC T1 pipeline

- Πολιτική επιλογής thread:
  - εναλλαγή μεταξύ διαθέσιμων threads σε κάθε κύκλο
  - προτεραιότητα στο least recently used thread (round-robin)
- Λόγοι μη διαθεσιμότητας (και μη επιλογής) ενός thread
  - long-latency εντολές (π.χ. branches, mult/div) οδηγούν στη μη-επιλογή του αντίστοιχου thread για όσους κύκλους διαρκούν
  - stalls λόγω cache misses
  - stalls λόγω structural hazards για μια non-pipelined δομή που χρησιμοποιείται ήδη από κάποιο άλλο thread (π.χ. divider)

UltraSPARC T1 pipeline

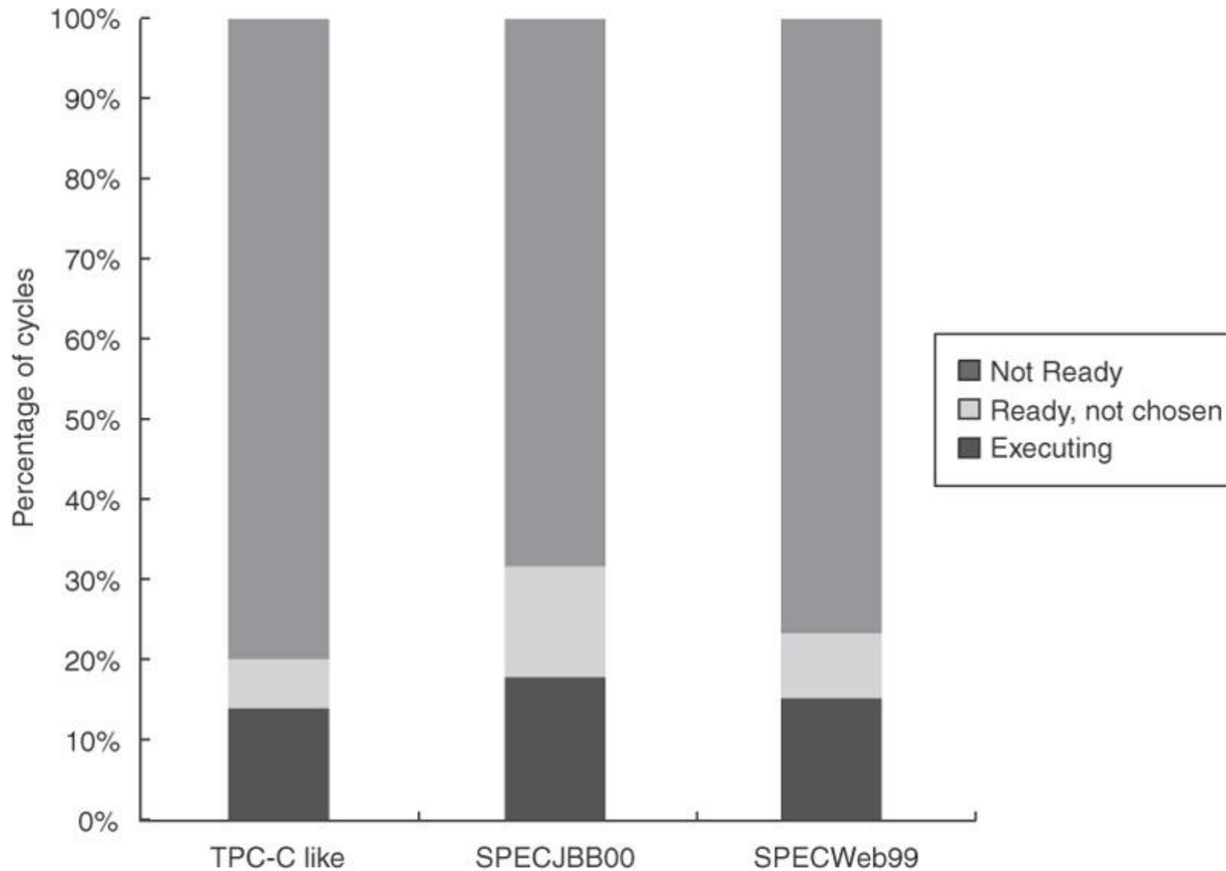


UltraSPARC T1 performance

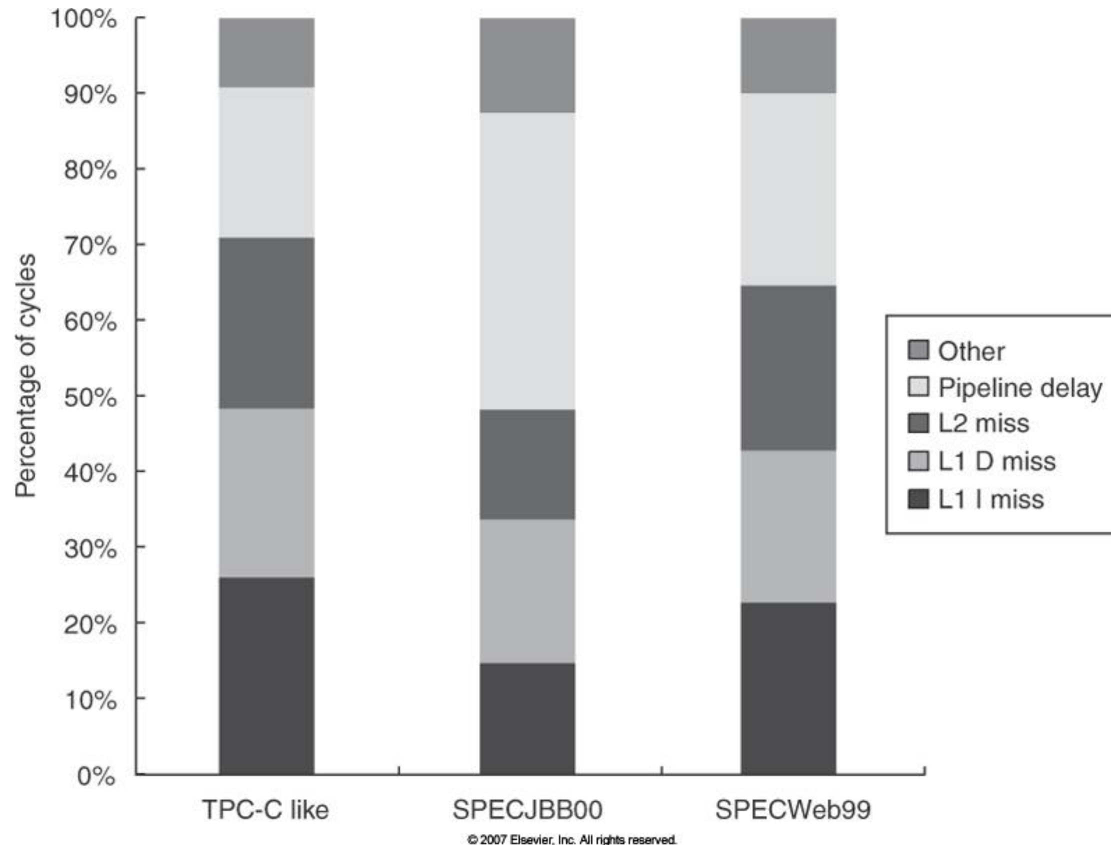
Benchmark	Per-thread CPI	Per core CPI	Effective CPI for eight cores	Effective IPC for eight cores
TPC-C	7.2	1.8	0.225	4.4
SPECJBB	5.6	1.40	0.175	5.7
SPECWeb99	6.6	1.65	0.206	4.8

- Fine-grained multithreading μεταξύ 4 threads
  → ιδανικό per-thread CPI = 4
- Ιδανικό per-core CPI = 1
- Effective CPI = per-core CPI / #cores
- Effective throughput: μεταξύ 56% και 71% του ιδανικού

Προφίλ εκτέλεσης ενός μέσου thread



Λόγοι για τη μη διαθεσιμότητα ενός thread



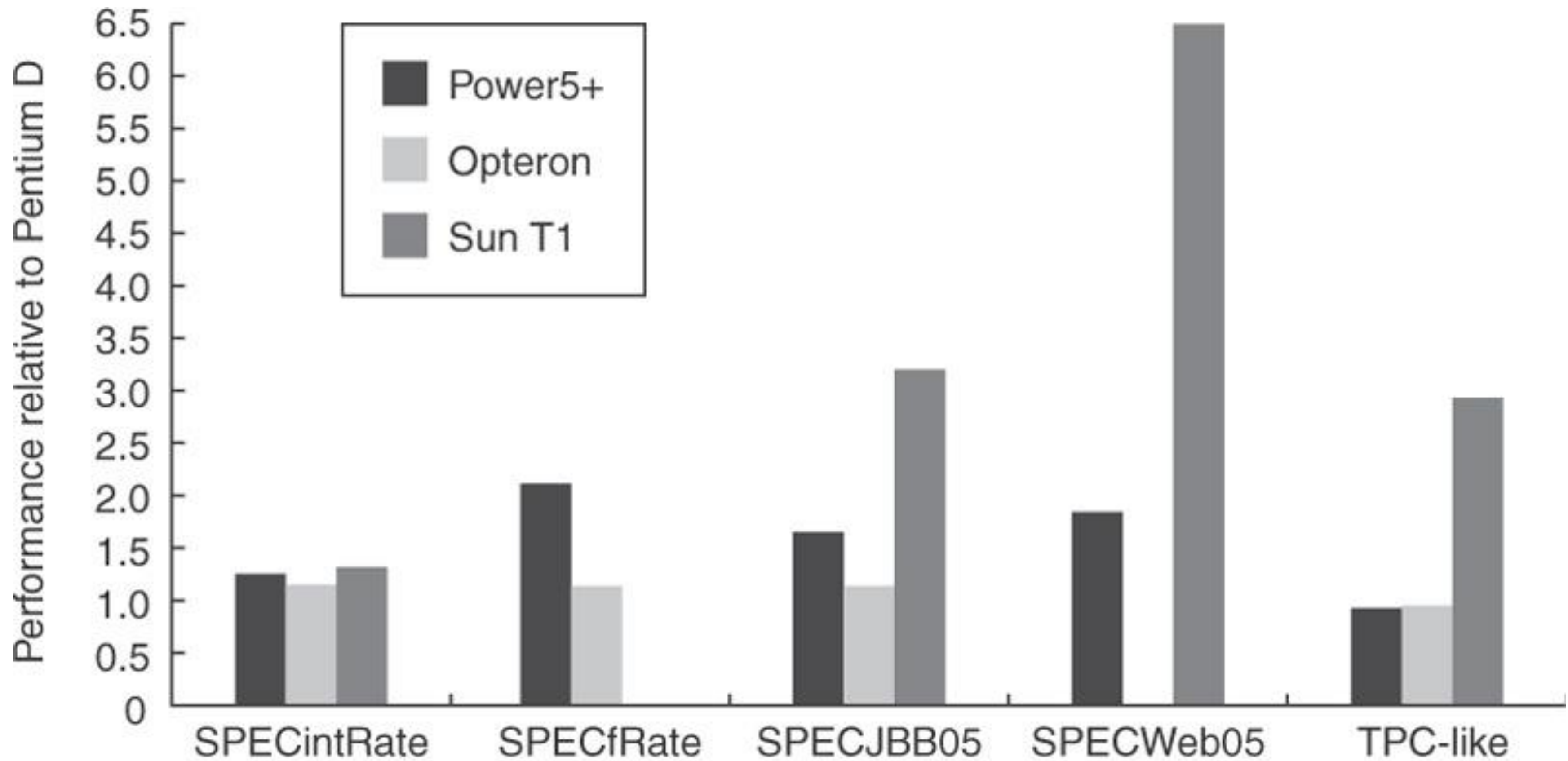
- Pipeline delay: long-latency εντολές όπως branches, loads, fp, int mult/div

«Crash-test» multicore επεξεργαστών (~2005)

Characteristic	SUN T1	AMD Opteron	Intel Pentium D	IBM Power5
Cores	8	2	2	2
Instruction issues per clock per core	1	3	3	4
Multithreading	Fine-grained	No	SMT	SMT
Caches L1 I/D in KB per core L2 per core/shared L3 (off-chip)	16/8 3 MB shared	64/64 1 MB/core	12K uops/16 1 MB/core	64/32 L2: 1.9 MB shared L3: 36 MB
Peak memory bandwidth (DDR2 DRAMs)	34.4 GB/sec	8.6 GB/sec	4.3 GB/sec	17.2 GB/sec
Peak MIPS FLOPS	9600 1200	7200 4800 (w. SSE)	9600 6400 (w. SSE)	7600 7600
Clock rate (GHz)	1.2	2.4	3.2	1.9
Transistor count (M)	300	233	230	276
Die size (mm^2)	379	199	206	389
Power (W)	79	110	130	125

- Βασικές διαφορές:
  - εκμετάλλευση ILP vs. TLP (Power5 → Opteron, Pentium D → T1)
  - floating point performance (Power5 → Opteron, Pentium D → T1)
  - memory bandwidth (T1 → Power5 → Opteron → Pentium D)
    - » επηρεάζει την απόδοση εφαρμογών με μεγάλο miss rate

«Crash-test» multicore επεξεργαστών

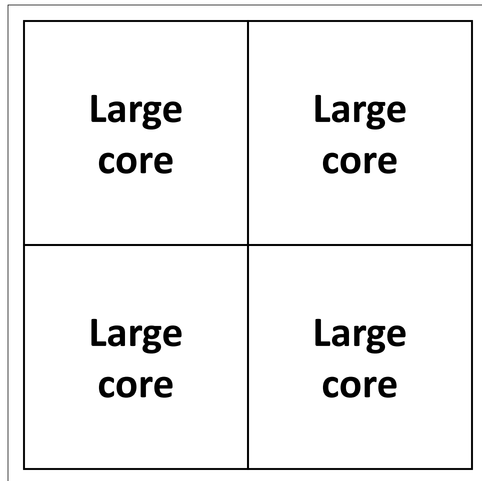


© 2007 Elsevier, Inc. All rights reserved.

«Crash-test» multicore επεξεργαστών (~2010)

	AMD Opteron 8439	IBM Power 7	Intel Xenon 7560	Sun T2
Transistors (M)	904	1200	2300	500
Power (W)	137	140	130	95
Max cores/chip	6	8	8	8
Mutlithreading	No	SMT	SMT	Fine-grained
Threads/ core	1	4	2	8
Instr. issue/clock	3 from 1 thread	6 from 1 thread	4 from 1 thread	2 from 2 threads
Clock rate (GHz)	2.8	4.1	2.7	1.6
Outermost cache	L3, 6MB, shared	L3, 32MB, shared or private/core	L3, 24MB shared	L2, 4MB, shared
Inclusion	No	Yes	Yes	Yes
Coherence protocol	MOESI	Extended MESI	MESIF	MOESI
Coherence implementation	Snooping	L3 Directory	L3 Directory	L2 Directory
Extended coherence support	Up to 8 processor chips (NUMA)	Up to 32 processor chips (UMA)	Up to 8 processor cores	Up to 2/4 chips (directly/external ASICs)

Tile-Large Approach



"Tile-Large"

- Tile a few large cores
- IBM Power 5, AMD Barcelona, Intel Core2Quad, Intel Nehalem

+ High performance on single thread, serial code sections

- Low throughput on parallel program portions

Tile-Small Approach

Small core	Small core	Small core	Small core
Small core	Small core	Small core	Small core
Small core	Small core	Small core	Small core
Small core	Small core	Small core	Small core

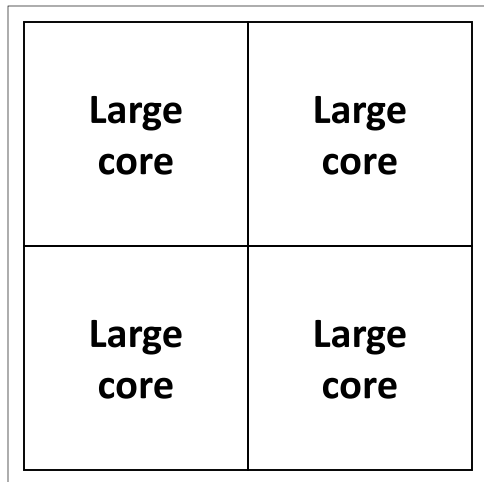
"Tile-Small"

- Tile many small cores
- Sun Niagara, Intel Larrabee, Tilera TILE (tile ultra-small)

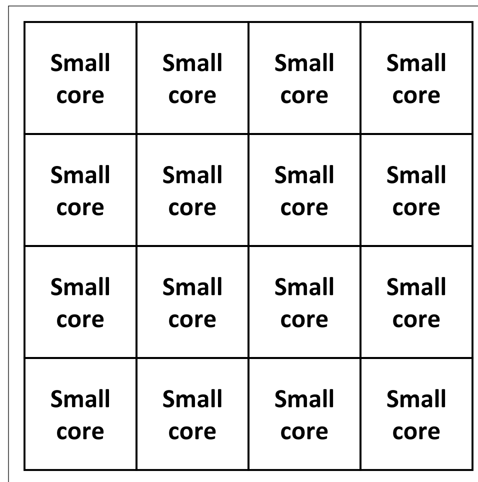
\+ High throughput on the parallel part (16 units)

\- Low performance on the serial part, single thread (1 unit)

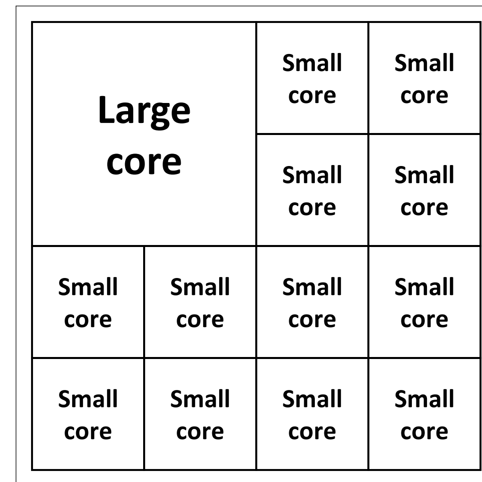
Asymmetric Chip Multiprocessor (ACMP)



"Tile-Large"



"Tile-Small"



ACMP

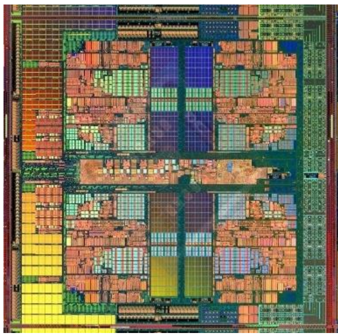
- Provide one large core and many small cores
- ARM big.LITTLE

\+ Accelerate serial part using the large core

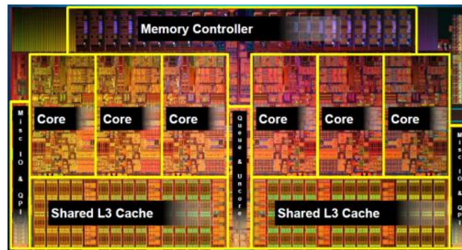
\+ Execute parallel part on small cores and large core for high throughput

Today: Many Cores on Chip

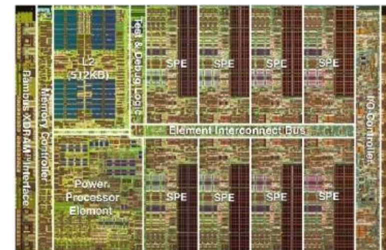
- Simpler and lower power than a single large core
- Large scale parallelism on chip



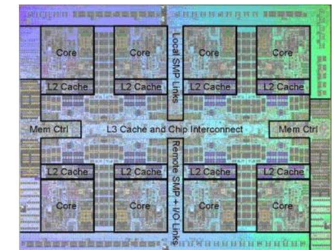
AMD Barcelona
4 cores



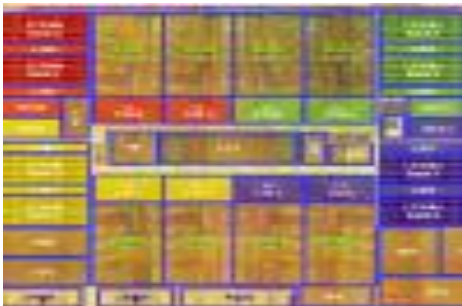
Intel Core i7
8 cores



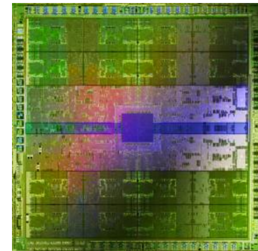
IBM Cell BE
8+1 cores



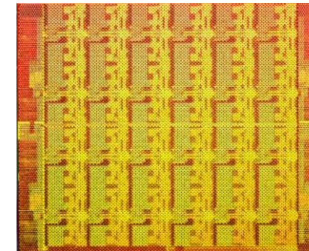
IBM POWER7
8 cores



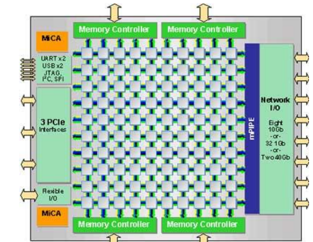
Sun Niagara II
8 cores



Nvidia Fermi
448 “cores”



Intel SCC
48 cores, networked



Tilera TILE Gx
100 cores, networked

Chips Today (2010-2014)

- Intel Nehalem (~2010)
  - slides by Krste Asanovic (Berkeley, CS152 - link)
- HotChips 2012 (HC24 http://www.hotchips.org/archives/hc24/)
  - Intel's 3rd generation processors Ivy Bridge (link)
  - AMD's Jaguar next generation low power x86 core (link)
  - Knight's Corner - Intel's MIC (link)
  - Power 7+ (link)
- HotChips 2013 (HC25 http://www.hotchips.org/archives/hc25/)
  - Power 8 (link)
  - Intel's 4th generation processors Haswell (link)
- HotChips 2014 (HC26 http://www.hotchips.org/archives/hc26/)
  - AMD's Kaveri APU (link)
  - AMD's Opteron A1100 (link)
  - Next generation SPARC Processor Cache Hierarchy (link)
  - Intel C2000 Atom Microserver (link)
  - NVIDIA's Denver Processor (link)
  - MIT Scorpio (link)
  - Powering the IoT (link)

Chips Today (2015-2017)

- HotChips 2015 (HC27 http://www.hotchips.org/archives/hc27/)
  - ARM Mali-T880GPU (link)
  - Intel's Knight Landing: 2nd Generation Xeon Phi Processor (link)
  - AMD's Carrizo APU (link)
  - Oracle's Sonoma Processor (link)
- HotChips 2016 (HC28 http://www.hotchips.org/archives/hc28/)
  - ARM v8-A (link)
  - Samsung Exynos-M1 (link)
  - NVIDIA Tegra SoC (link)
  - Oracle SPARC M7 (link)
  - Intel Skylake (link)
  - POWER9 (link)
- HotChips 2017 (HC29 http://www.hotchips.org/archives/hc29/)
  - Knights Mill: Intel Xeon Phi Processor for Machine Learning (link)
  - Celerity: An Open Source RISC-V Tiered Accelerator Fabric (link)
  - Graph Streaming Processor (GSP) A Next-Generation Computing Arch. (link)
  - The New Intel Xeon Processor Scalable Family (link)
  - And many more..

Chips Today (2018-)

- HotChips 2018 (HC30 https://www.hotchips.org/archives/2010s/hc30/)
  - Samsung M3 processor (link)
  - BROOM open-source OoO processor (link)
  - NVIDIA Xavier SoC (link)
  - ARM ML processor (link)
  - IBM Power9 Scale Up processor (link)
  - Xilinx DNN processor (link)
  - Vector Engine Processor of NEC's Aurora (link)
  - And many more…

Vector Αρχιτεκτονικές,
SIMD Extensions & GPUs

Πηγές/Βιβλιογραφία

- “Computer Architecture: A Quantitative Approach”, J. L. Hennessy, D. A. Patterson, Morgan Kaufmann Publishers, INC. 6th Edition, 2017
- Krste Asanovic, “Vectors & GPUs”, CS 152 Computer Architecture and Engineering, EECS Berkeley, 2018
  - https://inst.eecs.berkeley.edu/~cs152/sp18/lectures/L15-Vectors.pdf
  - https://inst.eecs.berkeley.edu/~cs152/sp18/lectures/L17-GPU.pdf
- Onur Mutlu, “SIMD Processors & GPUs”, Computer Architecture – ETH Zurich, 2017 (slides)
  - https://safari.ethz.ch/architecture/fall2017/lib/exe/fetch.php?media=onur-comparch-fall2017-lecture8-afterlecture.pdf
  - https://www.youtube.com/watch?v=6DqM1UpTZDM
  - https://safari.ethz.ch/architecture/fall2017/lib/exe/fetch.php?media=onur-comparch-fall2017-lecture9-afterlecture.pdf
  - https://www.youtube.com/watch?v=mgtlbEqn2dA

Προσεγγίσεις Αύξησης Απόδοσης

- **Παραλληλισμός σε επίπεδο εντολής (Instruction Level Parallelism – ILP)**
  - Εξαρτάται από τις πραγματικές εξαρτήσεις δεδομένων που υφίστανται ανάμεσα στις εντολές.

- **Παραλληλισμός σε επίπεδο νήματος (Thread-Level Parallelism – TLP)**
  - Αναπαρίσταται ρητά από τον προγραμματιστή, χρησιμοποιώντας πολλαπλά νήματα εκτέλεσης τα οποία είναι εκ κατασκευής παράλληλα.

- **Παραλληλισμός σε επίπεδο δεδομένων (Data-Level Parallelism – DLP)**
  - Αναπαρίσταται ρητά από τον προγραμματιστή ή δημιουργείται αυτόματα από τον μεταγλωττιστή.
  - Οι ίδιες εντολές επεξεργάζονται πολλαπλά δεδομένα ταυτόχρονα

Ταξινόμηση Παράλληλων Αρχιτεκτονικών

- *Single Instruction stream, Single Data stream (SISD)*
  - Uniprocessor.
- *Single Instruction stream, Multiple Data streams (SIMD)*
  - Πολλαπλοί επεξεργαστές, ίδιες εντολές, διαφορετικά δεδομένα (*data-level parallelism*).
- *Multiple Instruction streams, Single Data stream (MISD)*
  - Μέχρι σήμερα δεν έχει εμφανιστεί στην αγορά κάποιο τέτοιο σύστημα (είναι κυρίως για fault tolerance, π.χ. υπολογιστές που ελέγχουν πτήση αεροσκαφών).
- *Multiple Instruction streams, Multiple Data streams (MIMD)*
  - Ο κάθε επεξεργαστής εκτελεί τις δικές του εντολές και επεξεργάζεται τα δικά του δεδομένα. Πολλαπλά παράλληλα νήματα (thread-level parallelism).

[Mike Flynn, “Very high-speed computing systems”. Proc. of IEEE 54, 1966]

Παραλληλισμός σε επίπεδο Δεδομένων

- Προκύπτει από το γεγονός ότι οι ίδιες εντολές επεξεργάζονται πολλαπλά διαφορετικά δεδομένα ταυτόχρονα.
  - Πολλαπλές «επεξεργαστικές» μονάδες
- Παραδείγματα εφαρμογών με σημαντικό παραλληλισμό δεδομένων:
  - Επιστημονικές εφαρμογές (πράξεις με πίνακες)
  - Επεξεργασία εικόνας και ήχου
  - Αλγόριθμοι μηχανικής μάθησης

Παραλληλισμός σε επίπεδο Δεδομένων

- Γιατί Single Instruction Multiple Data (SIMD) αρχιτεκτονική?
- Μια εντολή → πολλαπλά δεδομένα → Καλύτερο energy-efficiency
  - Όταν υπάρχει DLP – εξαρτάται από την εφαρμογή, τον αλγόριθμο, προγραμματιστή, τον μεταγλωττιστή
  - Mobile devices
- Ο προγραμματιστής συνεχίζει να σκέφτεται (περίπου) ακολουθιακά

DLP Αρχιτεκτονικές

1. **Vector**

2. SIMD Extensions

3. GPUs

Vector Architecture

- Πρόκειται για αρχιτεκτονική που υποστηρίζει στο υλικό την εκτέλεση διανυσμάτων ή αλλιώς vectors
- Ένας vector (διάνυσμα) είναι ένας μονοδιάστατος πίνακας αριθμών
  - Εμφανίζονται σε πολλές εφαρμογές

```
for (i=0; i<n; i++)
   C[i] = A[i] + B[i];
```

- Ιστορικά
  - Εμφανίστηκαν στην δεκαετία του 1970
  - Κυριάρχησαν στον χώρο του supercomputing (1970-1990)
  - Σημαντικά πλεονεκτήματα για συγκεκριμένες εφαρμογές – επανέρχονται δυναμικά

Vector Architecture – Γενική Ιδέα

1. Φέρνει από την μνήμη ένα σύνολο δεδομένων και τα αποθηκεύει σε μεγάλα register files →Vector registers
2. Εκτελεί μία πράξη στο σύνολο όλων αυτών των δεδομένων
3. Γράφει τα αποτελέσματα πίσω στην μνήμη

- Οι καταχωρητές ελέγχονται από τον compiler και χρησιμοποιούντια για να:
  - Κρύψουν τον χρόνο πρόσβασης στην μνήμη
  - Εκμεταλλευτούν το memory bandwidth

Vector Instructions & Architecture Support

Vector Architecture – Functionality

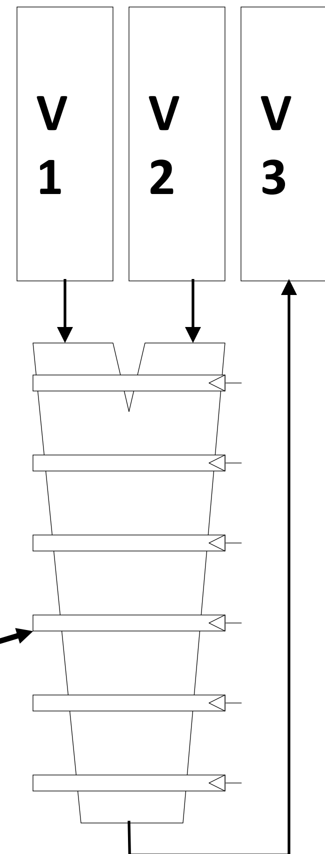
- Μια εντολή vector πραγματοποιεί μία πράξη σε κάθε στοιχείο του vector σε διαδοχικούς κύκλους
  - Pipelined functional units
  - Κάθε στάδιο του pipeline επεξεργάζεται ένα διαφορετικό στοιχείο

- Οι vector εντολές επιτρέπουν την υλοποίηση pipelines με περισσότερα στάδια (deep pipelines)
  - Δεν υπάρχουν εξαρτήσεις μέσα στους vectors
  - Δεν υπάρχει έλεγχος ροής μέσα στους vectors
  - Η γνώση των strides σχετικά με τις προσβάσεις στην μνήμη επιτρέπει αποτελεσματικό prefetching

Vector Architecture – Functionality

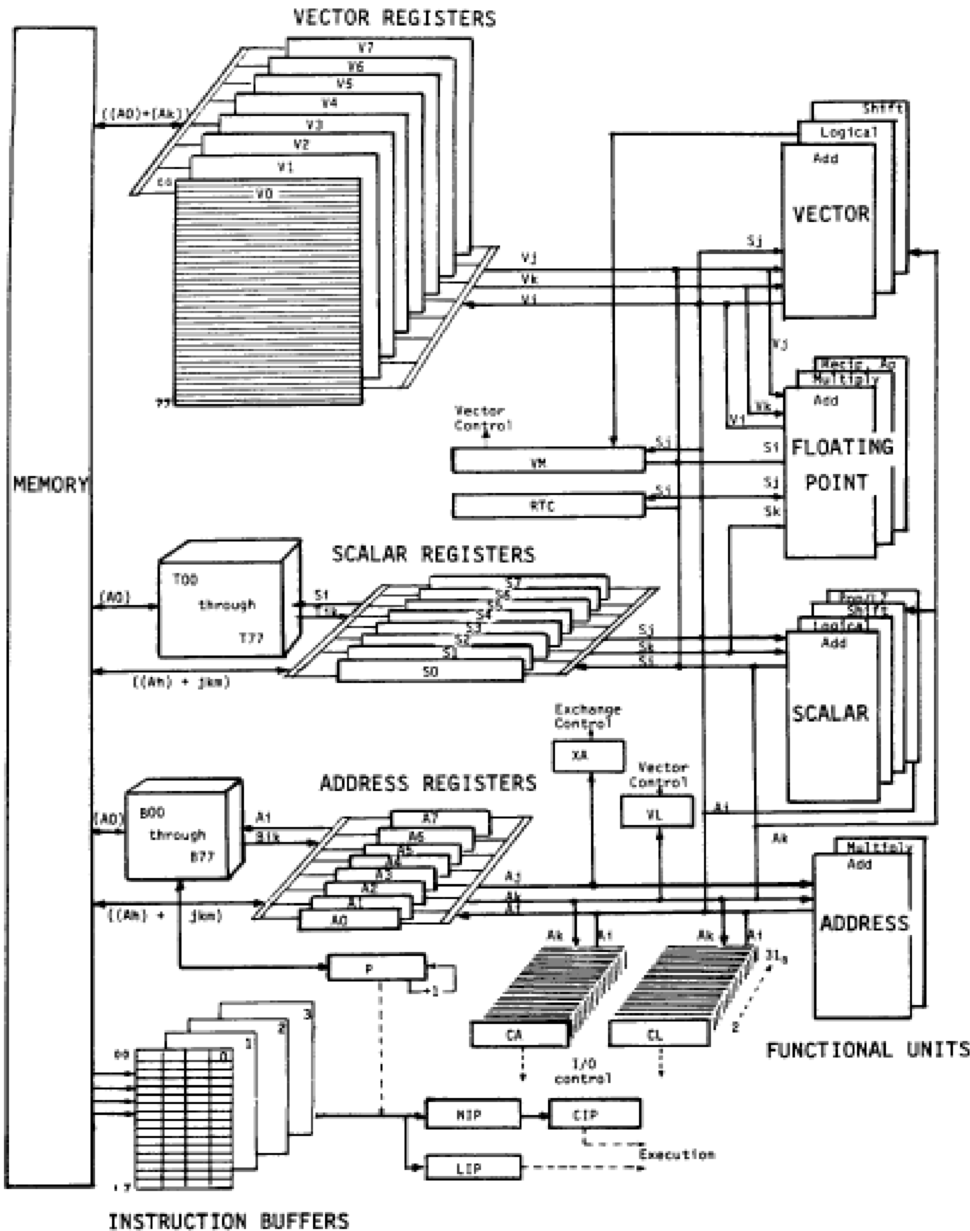
```
for (i=0; i<n; i++)
   C[i] = A[i] * B[i];
}
```

Six stage multiply pipeline



V1 * V2 → V3

Vector Architecture – Cray-1 (1978)



- Scalar Unit
  - Load/Store Architecture
- Vector Extensions
  - Vector Registers
  - Vector Instructions
- Highly pipelined functional units
- Interleaved memory system
  - Memory banks
- No data caches
- No virtual memory

Vector Architecture – Πλεονεκτήματα

- Όταν δεν υπάρχουν εξαρτήσεις μέσα στους vectors
  - Η pipeline τεχνική δουλεύει πολύ καλά
  - Επιτρέπει την ύπαρξη deep pipelines
- Κάθε εντολή αντιπροσωπεύει πολλή δουλειά
  - Απλούστερη λογική για instruction fetch και ανάγκη για λιγότερο memory bandwidth λόγω instructions
- Οι προσβάσεις στην μνήμη γίνονται με regular patterns
- Λιγότερα branch instructions

Vector Architecture – Μειονεκτήματα

- Δουλεύει καλά μόνο όταν υπάρχει διαθέσιμος παραλληλισμός δεδομένων στην εφαρμογή

- Μπορεί να δημιουργηθεί πρόβλημα λόγω memory bandwidth όταν:
  - Ο λόγος των εντολών υπολογισμού προς τις εντολές μνήμης δεν καλύπτει το κόστος της ανάγνωσης/εγγραφής των vectors από την μνήμη
  - Τα δεδομένα δεν είναι κατάλληλα αποθηκευμένα στην μνήμη (memory banks)

Vector Architecture – VMIPS

- RV64V (RISC-V base instructions + vector extensions)
  - Αρχιτεκτονική βασισμένη στον Cray-1
- Vector data registers
  - 32 registers, 64-bits wide each element
  - 16 read ports & 8 write ports
- Vector functional units
  - Fully pipelined
  - Data and control hazard detection
- Vector load/store unit
  - Fully pipelined
  - One word per clock cycle after initial latency
- Scalar registers
  - 31 general purpose registers + 32 floating point registers

Vector Architecture – Registers

- Vector data registers
- Vector control registers
  - VLEN, VMASK, VSTR
- VLEN (Vector Length Register)
  - Δηλώνει το μέγεθος του vector
  - Η μέγιστη τιμή του καθορίζεται από την αρχιτεκτονική
    - MVL – Maximum Vector Length
- VMASK (Vector Mask Register)
  - Δηλώνει τα στοιχεία του vector στα οποία εφαρμόζονται οι vector instructions
- VSTR (Vector Stride Register)
  - Δηλώνει την απόσταση των στοιχείων στην μνήμη για την δημιουργία ενός vector

DAXPY Παράδειγμα – Scalar Code

```
double a, X[N], Y[N];
for (i=0; i<32; i++) {
      Y[i] = a*X[i] + Y[i];
}
```

```
      fld      f0,a              # Load scalar a
      addi     x28,x5,#256       # Last address to load
Loop:
      fld      f1,0(x5)          # load X[i]
      fmul.d   f1,f1,f0          # a * X[i]
      fld      f2,0(x6)          # load Y[i]
      fadd.d   f2,f2,f1          # a * X[i] + Y[i]
      fsd      f2,0(x6)          # store into Y[i]
      addi     x5,x5,#8          # Increment index to X
      addi     x6,x6,#8          # Increment index to Y
      bne      x28,x5,Loop       # check if done
```

DAXPY Παράδειγμα – Vector Code

```
double a, X[N], Y[N];
for (i=0; i<32; i++) {
     Y[i] = a*X[i] + Y[i];
}
```

```
vsetdcfg 4*FP64   # Enable 4 DP FP vregs
fld      f0,a     # Load scalar a
vld      v0,x5    # Load vector X
vmul     v1,v0,f0 # Vector-scalar mult
vld      v2,x6    # Load vector Y
vadd     v3,v1,v2 # Vector-vector add
vst      v3,x6    # Store the sum
vdisable          # Disable vector regs
```

8 instructions for RV64V (vector code)

vs.

258 instructions for RV64G (scalar code)

Vector Execution Time

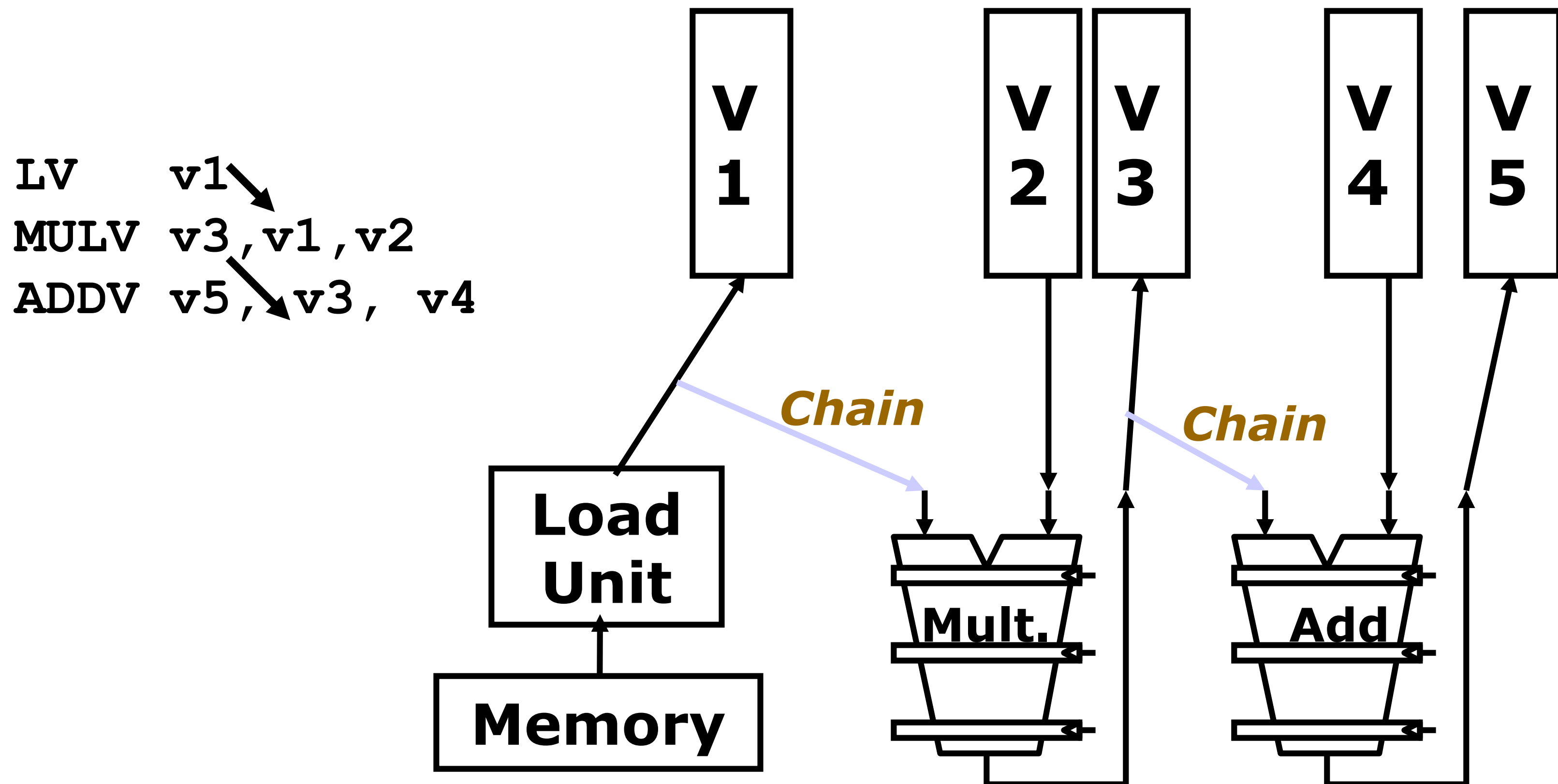
- Η επίδοση εξαρτάται από τρεις παράγοντες:
  - Μέγεθος των vectors
  - Δομικοί κίνδυνοι (structural hazards)
  - Εξαρτήσεις δεδομένων (data dependencies)

- Lanes
  - Πολλαπλά παράλληλα pipelines που παράγουν δύο ή περισσότερα αποτελέσματα σε κάθε κύκλο

- Convoy
  - Vector instructions που θα μπορούσαν να εκτελεστούν ταυτόχρονα
    - Χωρίς δομικούς κινδύνους

Vector Chaining & Chimes

- Ακολουθίες από read-after-write εξαρτήσεις δεδομένων τοποθετούνται στο ίδιο convoy και εκτελούνται μέσω της τεχνικής του ***chaining***



Vector Chaining & Chimes

- Chaining
  - Μια vector εντολή ξεκινάει να εκτελείται καθώς στοιχεία του vector source operand γίνονται διαθέσιμα

- Chime
  - Μονάδα χρόνου για την εκτέλεση ενός convoy
  - *m* convoys εκτελούνται σε *m chimes* για vector length *n*
  - Χρειάζεται *(m* x *n)* κύκλους για vector length *n*

Vector Chaining & Chimes – Παράδειγμα

```
vld     v0,x5       # Load vector X
vmul    v1,v0,f0    # Vector-scalar multiply
vld     v2,x6       # Load vector Y
vadd    v3,v1,v2    # Vector-vector add
vst     v3,x6       # Store the sum
```

Convoys:

1st chime:	vld	vmul
2nd chime:	vld	vadd
3rd chime:	vst	

- 3 chimes, 2 FP ops per result, cycles per FLOP = 1.5
- For 32 element vectors, requires 32 x 3 = 96 clock cycles

Vector Architecture – Challenges

- **Start up time** → Καθυστέρηση μέχρι να γεμίσει το pipeline
- **Execute more than vector elements per cycle?**
  - Multiple lanes execution
- **Vector length != Maximum Vector Length (MVL)?**
  - Vector Length Register + Strip mining
- **If statements + vector operations?**
  - Vector Mask Register + Predication
- **Memory system?**
  - Memory banks
- **Multiple dimensional matrices?**
  - Vector Stride Register
- **Sparse matrices?**
  - Scatter/gather operations
- **Programming a vector computer?**

Multiple Lanes Execution

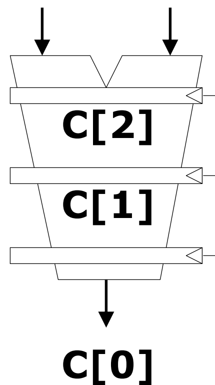
VADD A,B → C

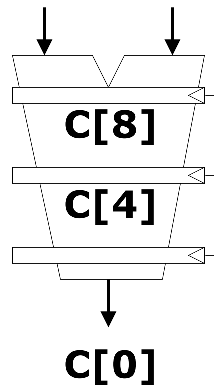
Execution using one pipelined functional unit

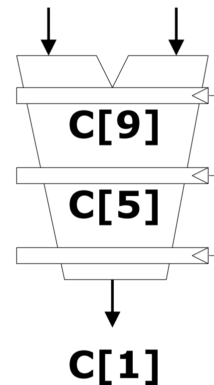
Execution using four pipelined functional units

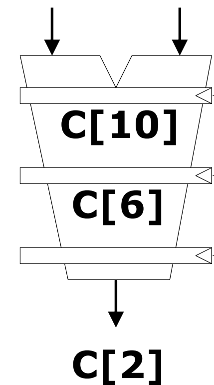
A[6]	**B[6]**
A[5]	**B[5]**
A[4]	**B[4]**
A[3]	**B[3]**

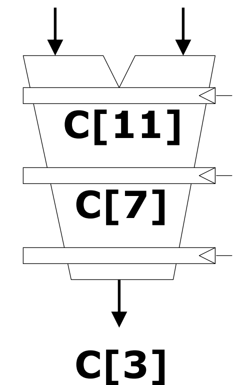
A[24]	**B[24]**	**A[25]**	**B[25]**	**A[26]**	**B[26]**	**A[27]**	**B[27]**
A[20]	**B[20]**	**A[21]**	**B[21]**	**A[22]**	**B[22]**	**A[23]**	**B[23]**
A[16]	**B[16]**	**A[17]**	**B[17]**	**A[18]**	**B[18]**	**A[19]**	**B[19]**
A[12]	**B[12]**	**A[13]**	**B[13]**	**A[14]**	**B[14]**	**A[15]**	**B[15]**





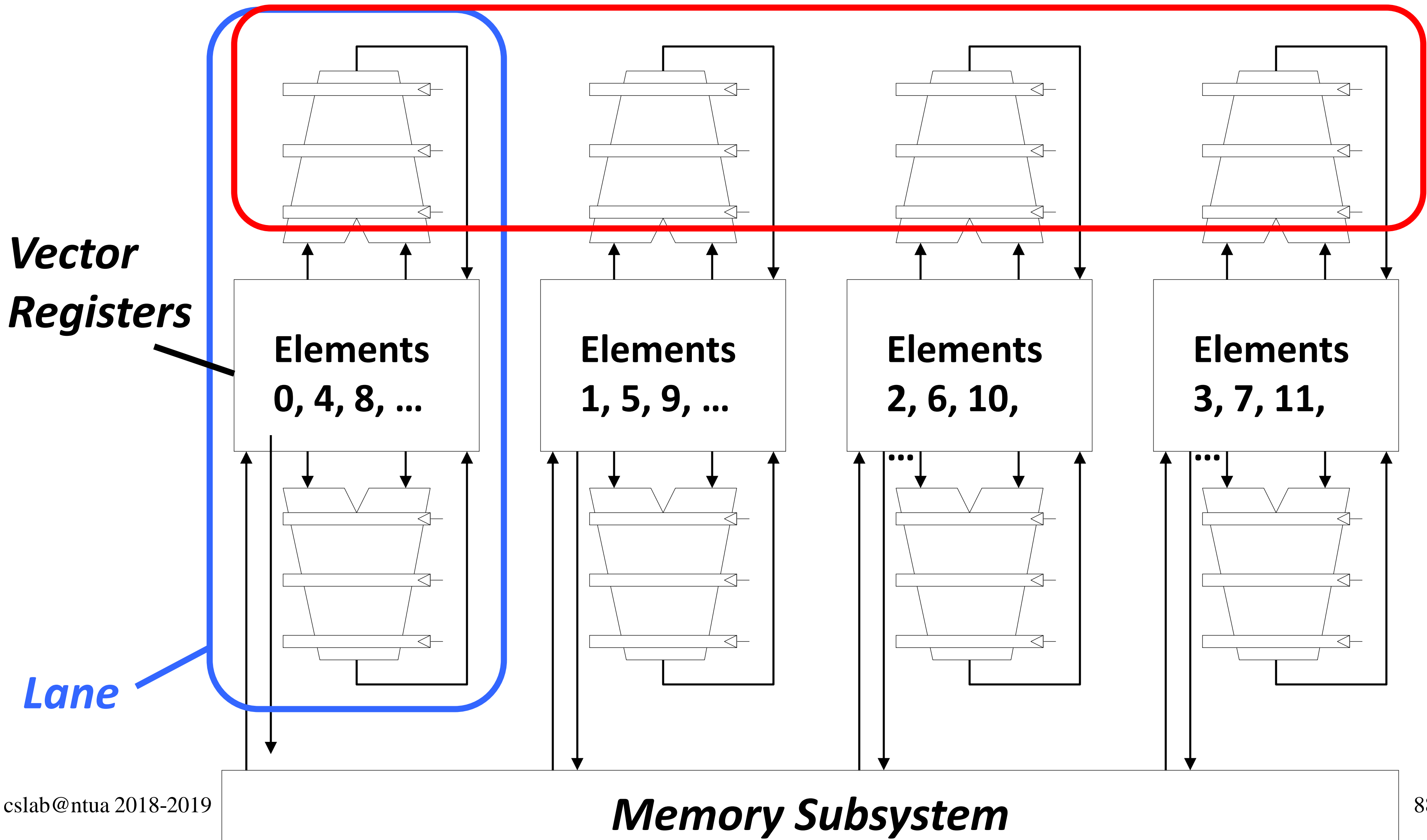






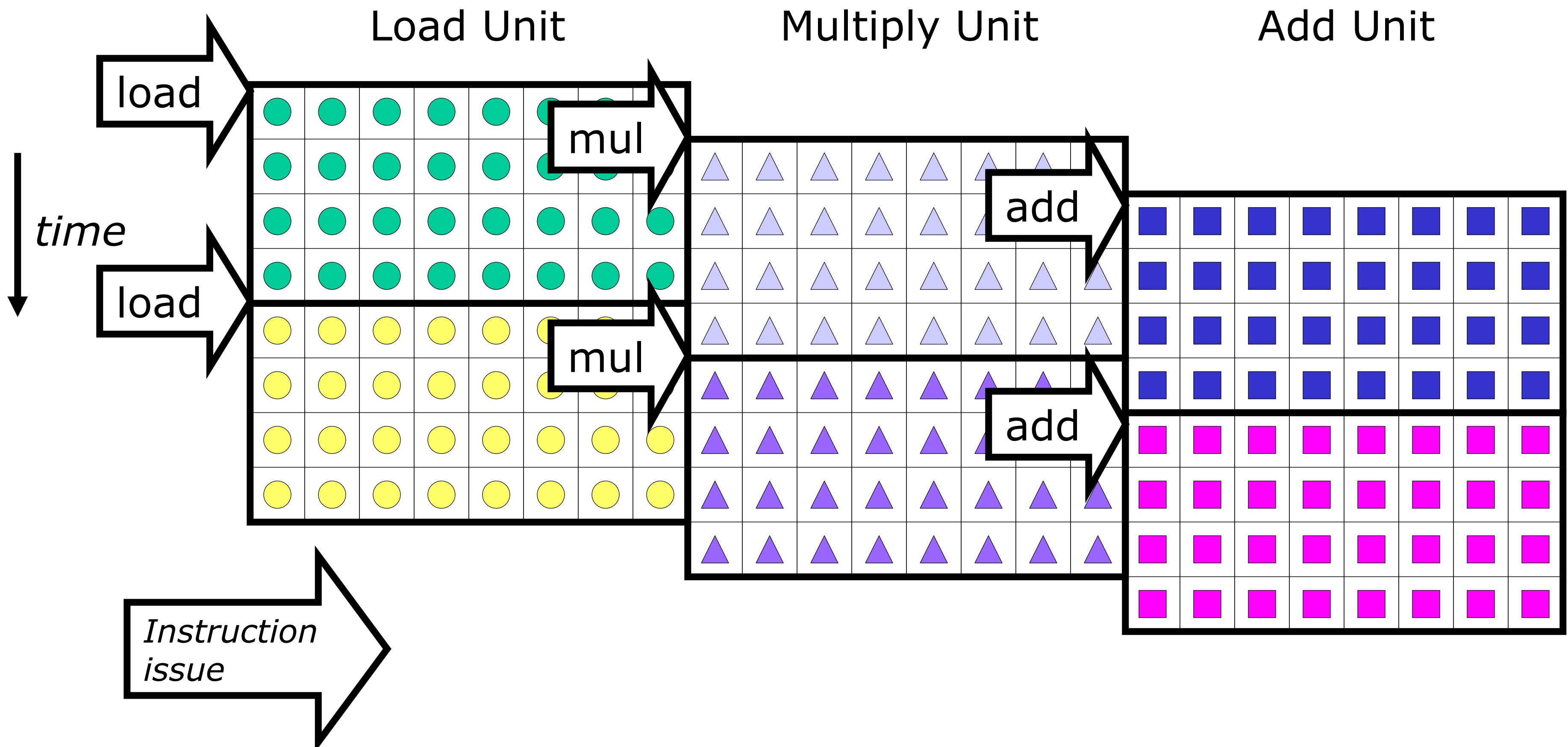
Multiple Lanes Execution

Το στοιχείο *n* του vector register A είναι "hardwired" στο στοιχείο *n* του vector register B → multiple HW lanes



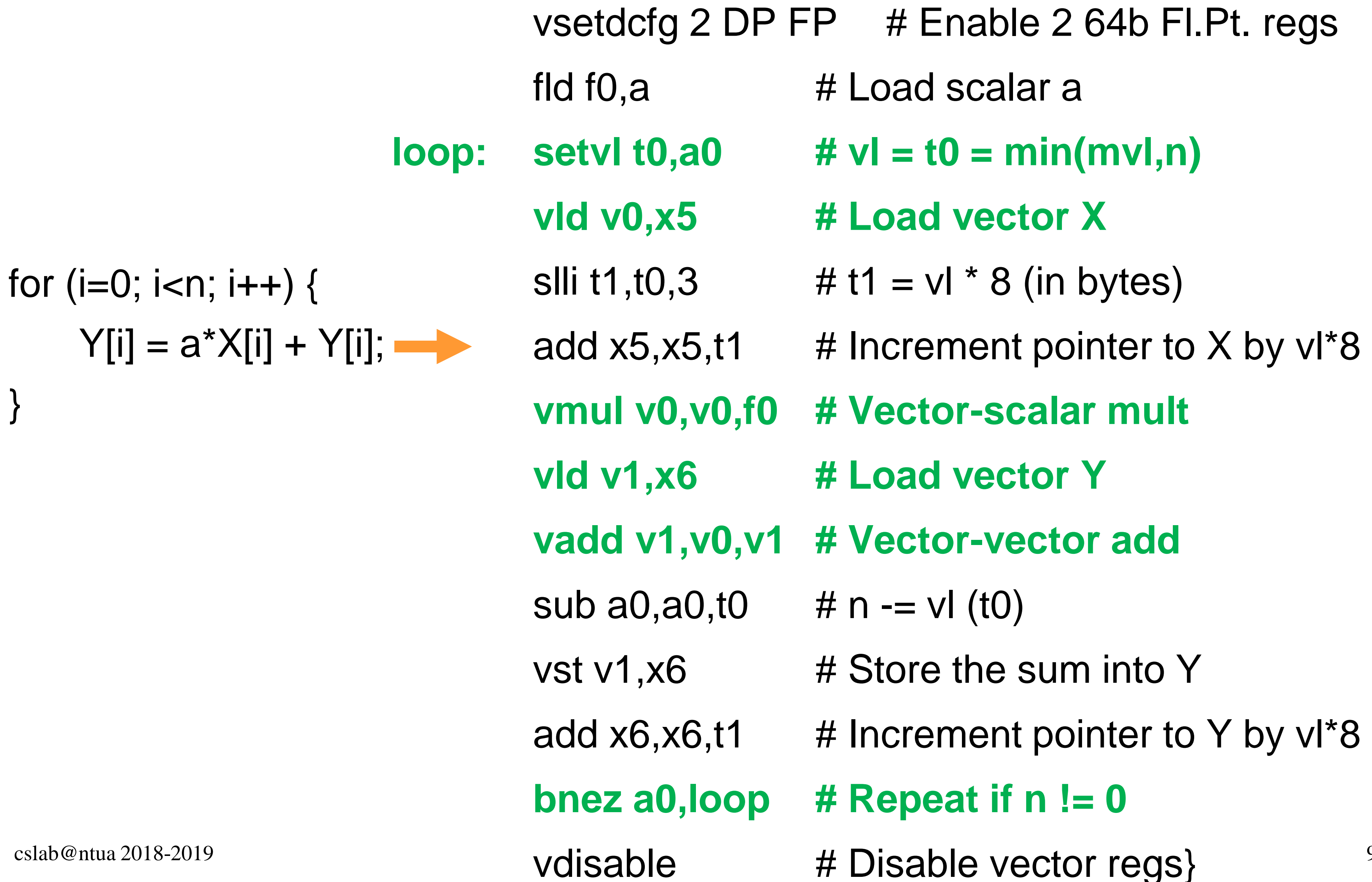
Vector Instruction Parallelism

- Can overlap execution of multiple vector instructions
  - example machine has 32 elements per vector register and 8 lanes



Complete 24 operations/cycle while issuing 1 short instruction/cycle

Vector Length Register – Strip-Mining Technique



Vector Mask Registers
Disable elements through predication

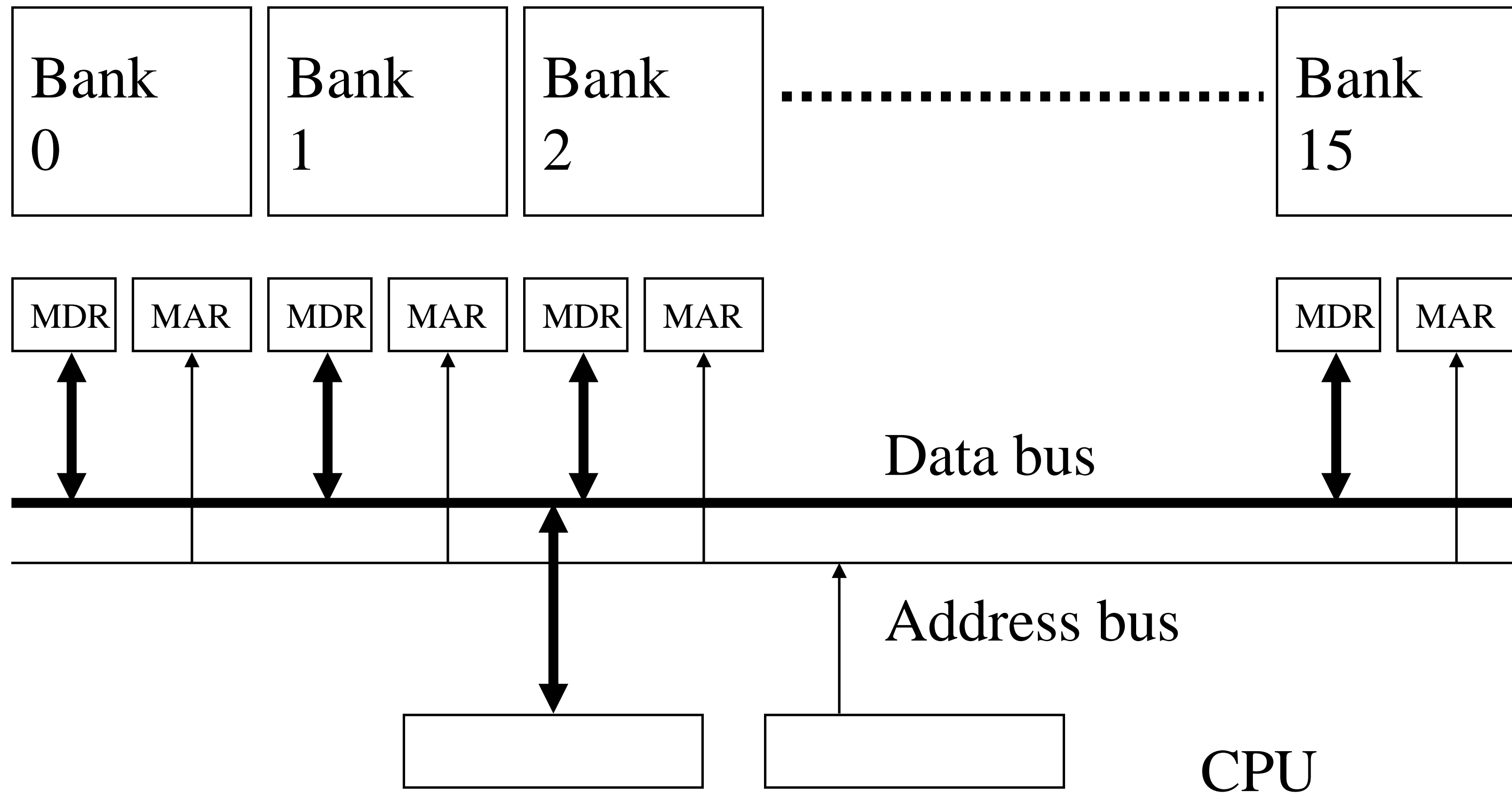
```
for (i = 0; i < 64; i=i+1) {
        if (X[i] != 0)
                X[i] = X[i] – Y[i];
}
```

```
vsetdcfg    2*FP64      # Enable 2 64b FP vector regs
vsetpcfgi   1           # Enable 1 predicate register
vld         v0,x5       # Load vector X into v0
vld         v1,x6       # Load vector Y into v1
fmv.d.x     f0,x0       # Put (FP) zero into f0
vpne        p0,v0,f0    # Set p0(i) to 1 if v0(i)!=f0
vsub        v0,v0,v1    # Subtract under vector mask
vst         v0,x5       # Store the result in X
vdisable                # Disable vector registers
vpdisable               # Disable predicate registers
```

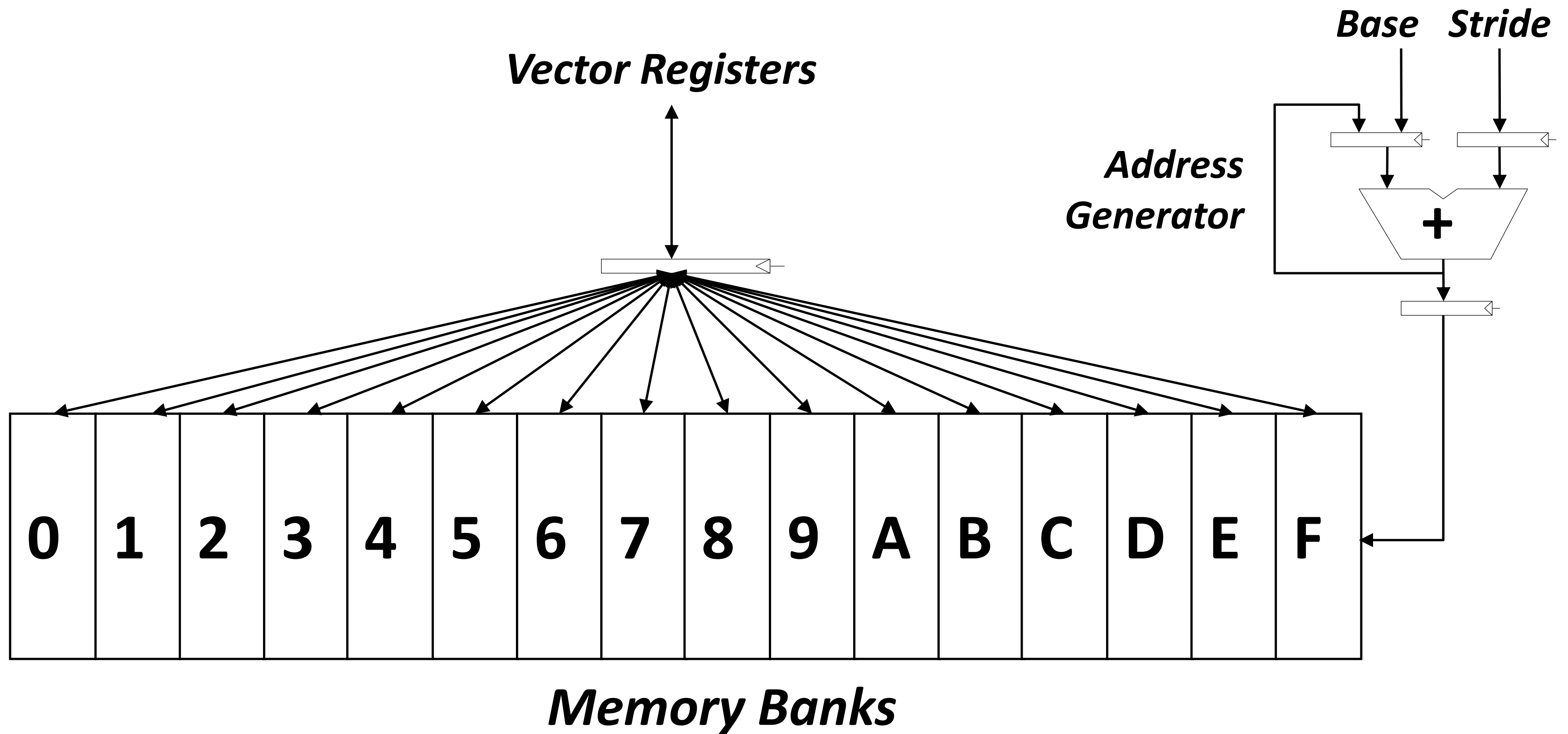
Memory Banks

- Το σύστημα μνήμης πρέπει να υποστηρίζει υψηλό bandwidth για vector loads & stores
- Προσέγγιση: Διαμοιρασμός των προσβάσεων μνήμης σε πολλαπλά banks
- Η μνήμη χωρίζεται σε banks τα οποία προσπελαύνονται ανεξάρτητα
  - Control bank addresses independently
  - Load or store non sequential words
  - Support multiple vector processors sharing the same memory
- Μπορεί να εξυπηρετήσει N παράλληλες προσπελάσεις αν όλες πηγαίνουν σε διαφορετικά banks

Memory Banks



Memory Banks



Stride Accesses

- Παράδειγμα:

```
for (i = 0; i < 100; i=i+1)
    for (j = 0; j < 100; j=j+1) {
        A[i][j] = 0.0;
        for (k = 0; k < 100; k=k+1)
            A[i][j] = A[i][j] + B[i][k] * D[k][j];
    }
```

- Vector Stride Register
  - Stride: απόσταση μεταξύ δύο στοιχείων για τον σχηματισμό vector
  - Ίσως προκύψουν bank conflicts

Scatter-Gather

- Παράδειγμα:

```
for (i = 0; i < n; i=i+1)
    A[K[i]] = A[K[i]] + C[M[i]];
```

- Index vector instructions

```
vsetdcfg   4*FP64        # 4 64b FP vector registers
vld        v0, x7        # Load K[]
vldx       v1, x5, v0    # Load A[K[]]
vld        v2, x28       # Load M[]
vldx       v3, x6, v2    # Load C[M[]]
vadd       v1, v1, v3    # Add them
vstx       v1, x5, v0    # Store A[K[]]
vdisable                 # Disable vector registers
```

Programming a Vector computer

- Ο compiler μπορεί να δώσει feedback στον προγραμματιστή
  - Για να εφαρμόσει vector optimizations

- Ο προγραμματιστής μπορεί να δώσει hints στον compiler
  - Ποια κομμάτια κώδικα να γίνουν vectorized

DLP Αρχιτεκτονικές

1. Vector

2. **SIMD Extensions**

3. GPUs

SIMD Extensions

- Πολλές εφαρμογές πολυμέσων επεξεργάζονται δεδομένα που έχουν μέγεθος μικρότερο από το μέγεθος λέξης (π.χ. 32 bits) για το οποίο ο επεξεργαστής είναι βελτιστοποιημένος
- Παραδείγματα:
  - Τα pixels γραφικών αναπαριστώνται συνήθως με 8 bits για κάθε κύριο χρώμα + 8 bits για transparency
  - Τα δείγματα ήχου αναπαριστώνται συνήθως με 8 ή 16 bits

- SIMD Extensions
  - Διαμοιρασμός των functional units για την ταυτόχρονη επεξεργασία στοιχείων μικρών vectors
  - Παράδειγμα: “partitioned” adder

SIMD Extensions vs. Vectors

- SIMD extensions operate on *small vectors*
  - Πολύ μικρότερα register files
  - Λιγότερη ανάγκη για υψηλό memory bandwidth

- Περιορισμοί των SIMD Extensions:
  - Ο αριθμός των data operands αντικατοπτρίζεται στο op-code
    - Το instruction set γίνεται περισσότερο περίπλοκο
  - Δεν υπάρχει Vector Length Register
  - Δεν υποστηρίζονται περίπλοκοι τρόποι διευθυνσιοδότησης (strided, scatter/gather)
  - Δεν υπάρχει Vector Mask Register

SIMD Extensions – Υλοποιήσεις

- Intel MMX (1996)
  - Eight 8-bit integer ops or four 16-bit integer ops
  - Peleg and Weiser, “MMX Technology Extension to the Intel Architecture”, IEEE Micro, 1996
- Streaming SIMD Extensions (SSE) (1999)
  - Eight 16-bit integer ops
  - Four 32-bit integer/fp ops or two 64-bit integer/fp ops
- Advanced Vector Extensions (2010)
  - Four 64-bit integer/fp ops
- AVX-512 (2017)
  - Eight 64-bit integer/fp ops
- Operands must be consecutive and aligned memory locations

SIMD Extensions – Παράδειγμα

```
for (i=0; i<n; i++) {
    Y[i] = a*X[i] + Y[i];
}
```

```
        fld       f0,a            # Load scalar a
        splat.4D  f0,f0           # Make 4 copies of a
        addi      x28,x5,#256     # Last addr. to load
Loop:
        fld.4D    f1,0(x5)        # Load X[i] ... X[i+3]
        fmul.4D   f1,f1,f0        # a x X[i] ... a x X[i+3]
        fld.4D    f2,0(x6)        # Load Y[i] ... Y[i+3]
        fadd.4D   f2,f2,f1        # a x X[i]+Y[i] …
                                  # … a x X[i+3]+Y[i+3]
        fsd.4D    f2,0(x6)        # Store Y[i]... Y[i+3]
        addi      x5,x5,#32       # Increment index to X
        addi      x6,x6,#32       # Increment index to Y
        bne       x28,x5,Loop     # Check if done
```

DLP Αρχιτεκτονικές

1. Vector

2. SIMD Extensions

3. **GPUs**

Graphical Processing Units

- Επιταχυντές για την επεξεργασία γραφικών
  - Υψηλός παραλληλισμός σε εφαρμογές γραφικών

- GP-GPUs
  - **G**eneral-**P**urpose computation on **G**raphics **P**rocessing **U**nits
  - Χρησιμοποιούνται ευρέως για την επιτάχυνση data και compute intensive εφαρμογών

- Εφαρμογές ιδανικές για GPGPUs
  - Υψηλός παραλληλισμός
  - Υψηλό arithmetic intensity
  - Μεγάλα data sets

Graphical Processing Units

- Ετερογενές μοντέλο εκτέλεσης
  - H CPU είναι ο *Host*
  - H GPU είναι το *device*
  - Επικοινωνία μέσω PCIe bus
- Μοντέλο προγραμματισμού
  - C-like γλώσσα προγραμματισμού για GPU
  - Βασίζεται στην έννοια των “threads”
  - **Single Instruction, Multiple threads (SIMT)**
    - Multi-threaded προγραμματιστικό μοντέλο
    - Διαφορετικό από SIMD που χρησιμοποιεί data parallel προγραμματιστικό μοντέλο
    - Συνδυάζει SIMD και Multithreading

GPUs & Flynn's Taxonomy

- Οι GPUs αποτελούνται από πολλαπλούς επεξεργαστές
  - Κάθε ένας επεξεργαστής αποτελείται από ένα multithreaded SIMD pipeline
- Το pipeline εκτέλεσης των εντολών λειτουργεί σαν ένα SIMD pipeline

- Όμως, ο προγραμματισμός γίνεται με την έννοια των threads
  - Όχι με την έννοια των SIMD εντολών

- Κάθε thread εκτελεί την ίδια εντολή αλλά επεξεργάζεται διαφορετικά δεδομένα
- Κάθε thread έχει το δική του κατάσταση και μπορεί να εκτελεστεί ανεξάρτητα

GPUs & Flynn's Taxonomy (2)

- SIMD Extensions
  - Το SIMD υλικό αξιοποιείται μέσα από:
    - *data parallel* προγραμματιστικό μοντέλο
      - Ο προγραμματιστής (ή ο μεταγλωττιστής) χρησιμοποιεί τις SIMD εντολές για να εκτελέσει την ίδια εντολή σε πολλά δεδομένα
    - Το οποίο εκτελείται μέσα από ένα SIMD μοντέλο εκτέλεσης
- GPUs
  - Το SIMD υλικό αξιοποιείται μέσα από:
    - Multithreaded προγραμματιστικό μοντέλο
      - Ο προγραμματιστής (ή ο μεταγλωττιστής) παράγει σειριακό κώδικα και δημιουργεί νήματα για να εκτελέσει τον ίδιο κώδικα σε πολλά δεδομένα
    - Το οποίο εκτελείται μέσα από ένα SIMD μοντέλο εκτέλεσης

SIMD vs. SIMT Μοντέλο Εκτέλεσης Υλικού

- Μοντέλο Εκτέλεσης Υλικού
  - Καθορίζει πως εκτελείται ένας κώδικας στο υλικό

- SIMD – Single Instruction Multiple Data
  - Μία ροή από διαδοχικές SIMD εντολές
  - Κάθε SIMD εντολή καθορίζει πολλά δεδομένα προς επεξεργασία

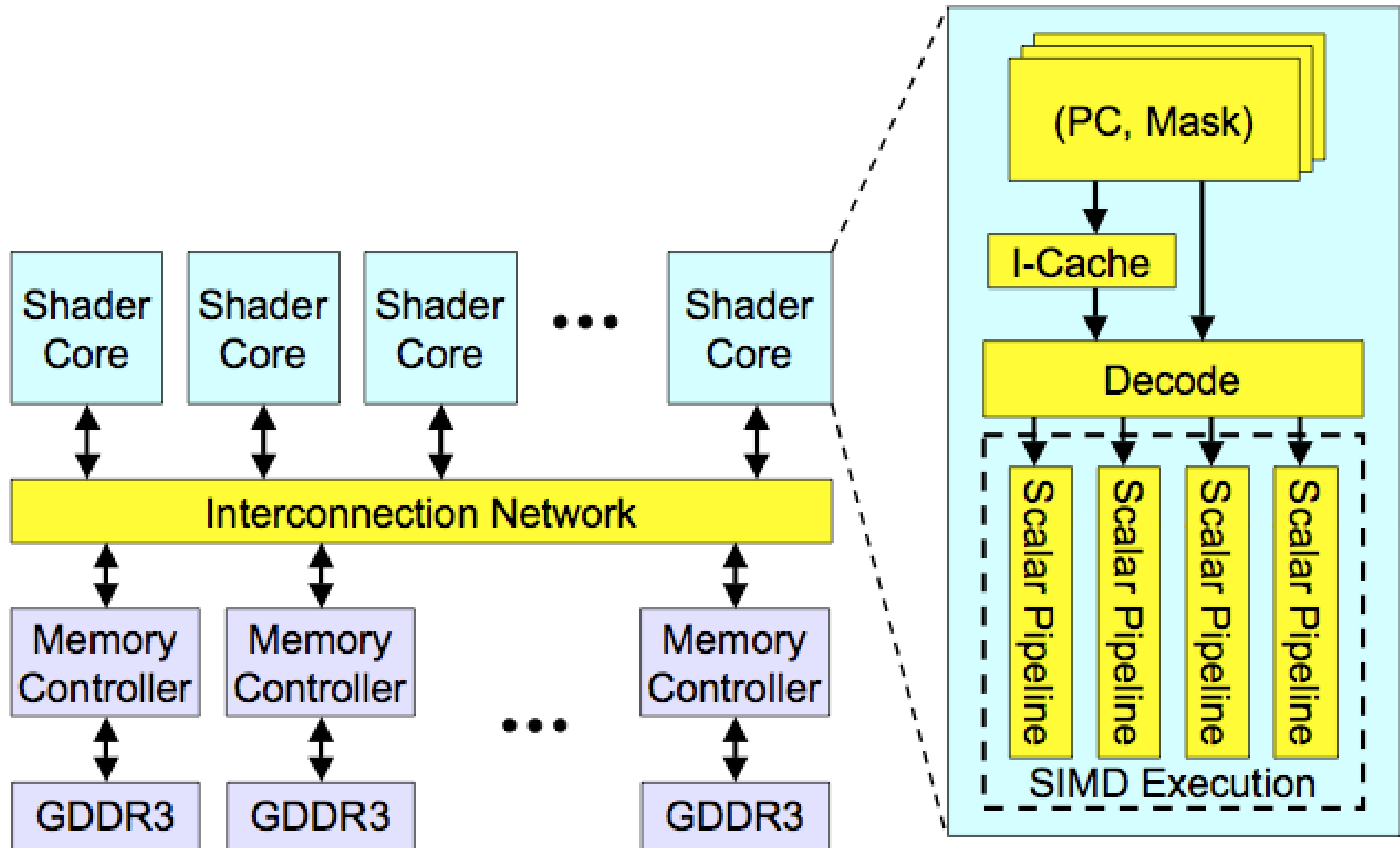
- SIMT – Single Instruction Multiple Threads
  - Πολλές ροές από scalar εντολές (μία για κάθε νήμα εκτέλεσης)
  - Πολλά νήματα ομαδοποιούνται *δυναμικά από το hardware* για να εκτελέσουν την ίδια ροή σε διαφορετικά δεδομένα

Πλεονεκτήματα SIMT Μοντέλου Εκτέλεσης

- Το υλικό μεταχειρίζεται κάθε thread σαν ανεξάρτητη οντότητα
  - Μπορεί να εκτελέσει κάθε thread ανεξάρτητα
  - Κερδίζοντας τα οφέλη του MIMD μοντέλου εκτέλεσης

- Το υλικό μπορεί να σχηματίσει δυναμικά groups από threads
  - Που εκτελούν την ίδια εντολή
  - Κερδίζοντας τα οφέλη του SIMD μοντέλου εκτέλεσης

GPU Αρχιτεκτονική



Threads and Blocks

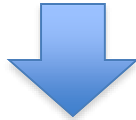
- Ένα *thread* συνδέεται με την επεξεργασία ενός υποσυνόλου του *data set (data elements)*
- Τα *threads* οργανώνονται σε *blocks*
- Τα *blocks* οργανώνονται σε ένα *grid*

- Το υλικό (hardware) της GPU διαχειρίζεται τα νήματα
  - Όχι οι εφαρμογές ή το λειτουργικό σύστημα
  - Multithreading SIMD execution

Παράδειγμα

CPU code

```
for (i = 0; i < 8192; ++i) {
    A[i] = B[i] * C[i];
}
```



CUDA code

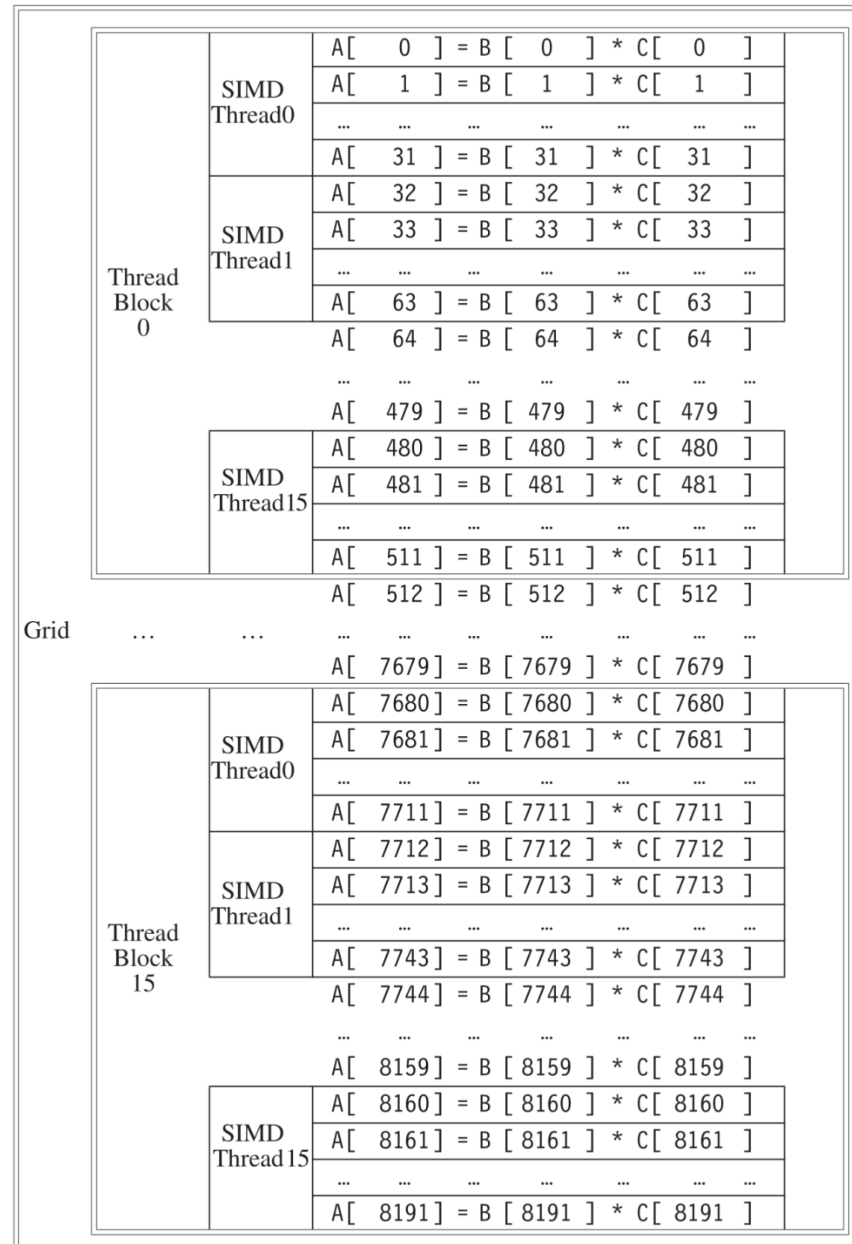
```
// there are 8192 threads
__global__ void KernelFunction(…) {
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  int varB = B[tid];
  int varC = C[tid];
  A[tid] = varB + varC;
}
```

Παράδειγμα – Threads and Blocks

- Code that works over all elements is the **grid**
- **Thread blocks** break this down into manageable sizes
  - 512 threads per block (defined by the programmer)
- **Groups of 32 threads** combined into a **SIMD thread** or **“warp”**
- Grid size = 16 thread blocks
  - 8192 elements / 512 threads per block
- Block is analogous to a strip-mined vector loop with vector length of 32

- A thread block is assigned to a **multithreaded SIMD processor** by the thread block scheduler
- **Current-generation GPUs have tens of multithreaded SIMD processors**

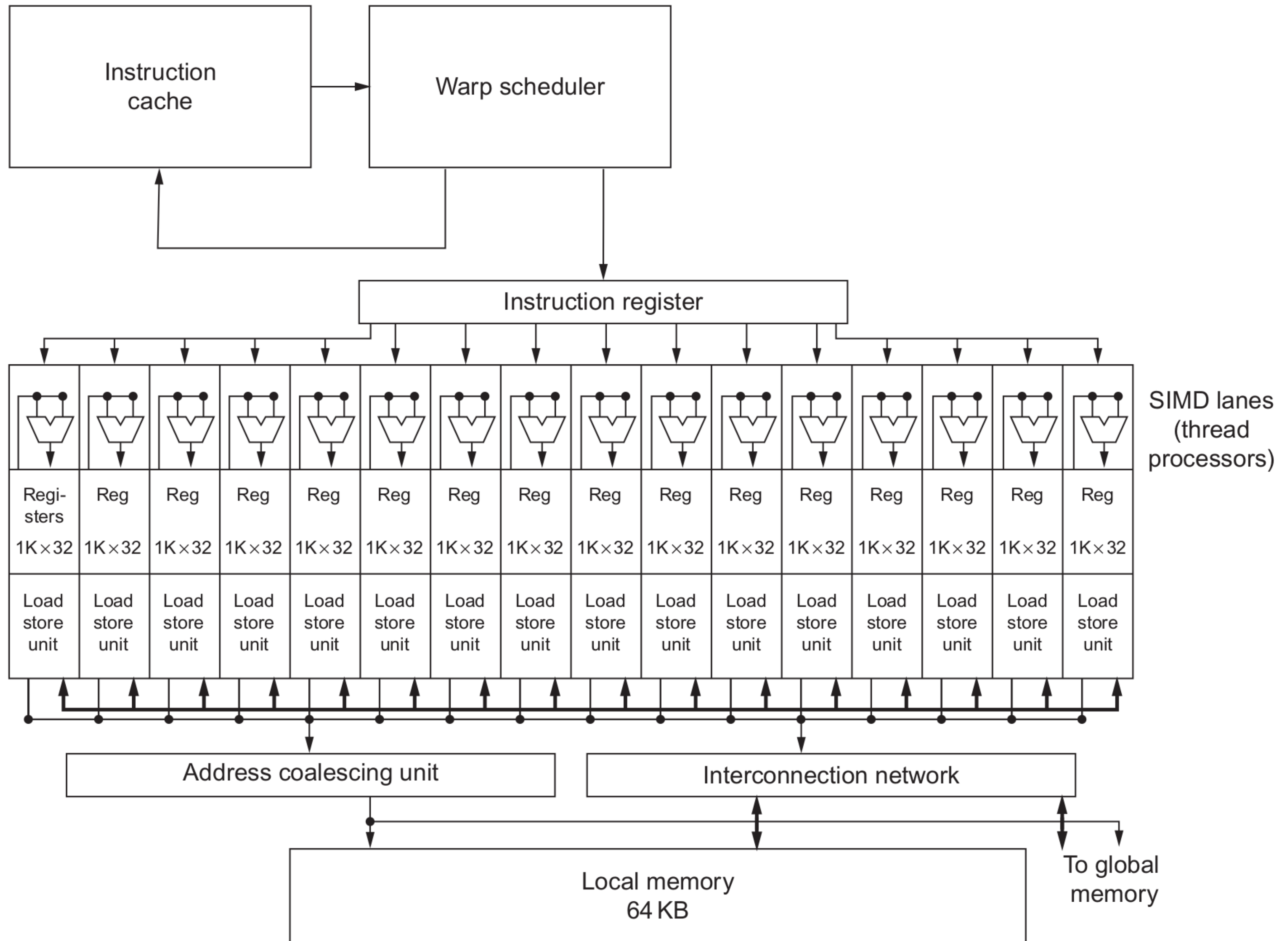
Παράδειγμα – Threads and Blocks



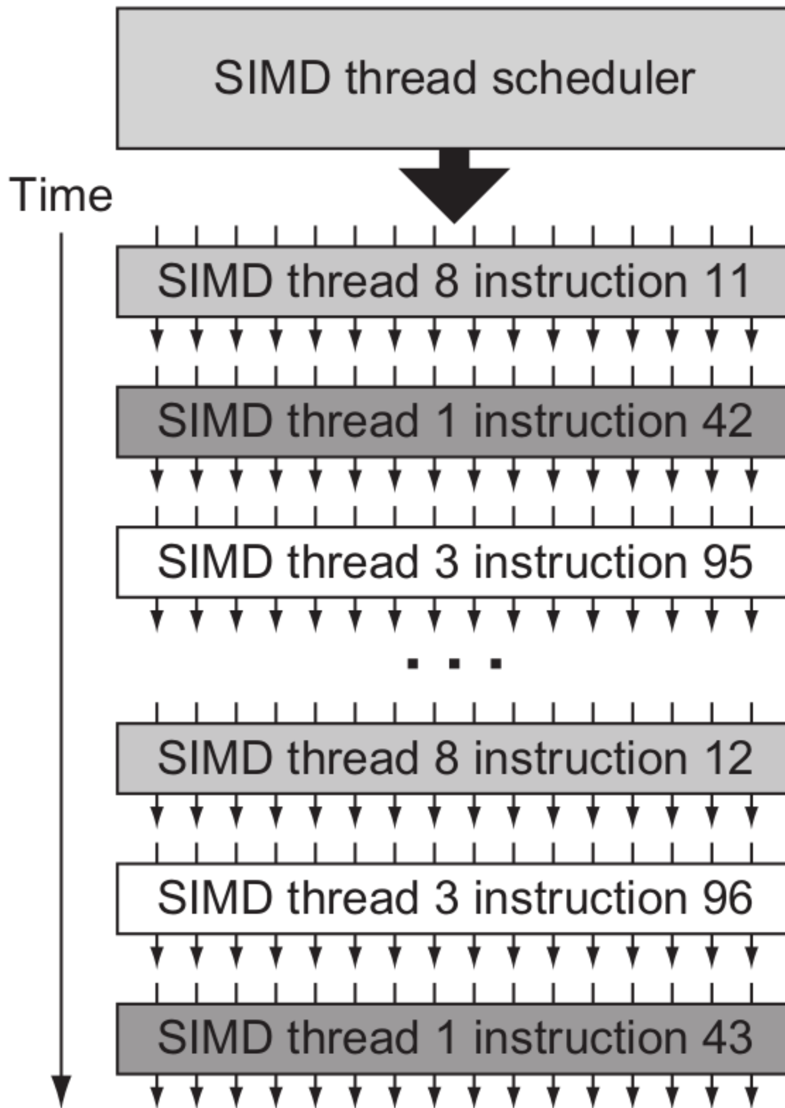
Παράδειγμα – GPU αρχιτεκτονική

- **Groups of 32 threads** combined into a **SIMD thread** or **"warp"**
  - Mapped to 16 (or 32) physical lanes
- Up to 32 warps are scheduled on a single SIMD processor
  - **Each warp has its own PC**
  - Each thread in a warp has its own register set (depends on the architecture & limits the number of warps per SIMD processor)
  - Thread scheduler uses scoreboard to dispatch warps
  - By definition, no data dependencies between warps
  - Dispatch warps into pipeline, hide memory latency
- Thread block scheduler schedules blocks to SIMD processors
- Within each SIMD processor:
  - **Wide and shallow** pipelined functional units compared to vector processors

Παράδειγμα – GPU αρχιτεκτονική



Warps are multithreaded on the SIMD core



- Warp == SIMD thread
- One warp is a single thread in the hardware
- Multiple warps are interleaved in execution on a single SIMD processor to hide latencies (memory and functional unit)
- A single thread block can contain multiple warps, all mapped to single SIMD processor
- Can have multiple thread blocks executing on one SIMD processor

NVIDIA Instruction Set Architecture

- ISA is an abstraction of the hardware instruction set
  - **“Parallel Thread Execution (PTX)”**
    - opcode.type d,a,b,c;
  - Χρησιμοποιεί virtual registers
  - Η μετάφραση σε κώδικα μηχανής πραγματοποιείται από το software
  - Παράδειγμα:

```
shl.s32     R8, blockIdx, 9          ; Thread Block ID * Block size (512 or 29)
add.s32     R8, R8, threadIdx        ; R8 = i = my CUDA thread ID
ld.global.f64     RD0, [X+R8]        ; RD0 = X[i]
ld.global.f64     RD2, [Y+R8]        ; RD2 = Y[i]
mul.f64 R0D, RD0, RD4                ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 R0D, RD0, RD2                ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0            ; Y[i] = sum (X[i]*a + Y[i])
```

Conditional Branching

- Like vector architectures, GPU branch hardware uses internal masks
- Also uses
  - Branch synchronization stack
    - Entries consist of masks for each SIMD lane
    - I.e. which threads commit their results (all threads execute)
  - Instruction markers to manage when a branch diverges into multiple execution paths
    - Push on divergent branch
  - …and when paths converge
    - Act as barriers
    - Pops stack
- Per-thread-lane 1-bit predicate register, specified by programmer

GPU Memory Structures

- **Shallow memory hierarchy**
  - **Multithreading → hides memory latency**
- Each SIMD Lane has private section of off-chip DRAM
  - **“Private memory”** (“local memory” in NVIDIA’s terminology)
  - Contains stack frame, spilling registers, and private variables
- Each multithreaded SIMD processor also has local memory
  - **“local memory”** (“shared memory” in NVIDIA’s terminology)
  - Scratchpad memory → managed explicitly by the programmer
  - Shared by SIMD lanes / threads within a block
- Memory shared by SIMD processors is GPU Memory
  - **“GPU memory”** (“global memory” in NVIDIA’s terminology)
  - Host can read and write GPU memory

NVIDIA's Pascal Architecture Innovations

- Each SIMD processor has:
  - Two SIMD thread (warp) schedulers, two instruction dispatch units
  - Two sets of:
    - 16 SIMD lanes (SIMD width=32, chime=2 cycles)
    - 16 load-store units,
    - 8 special function units
  - Two threads of SIMD instructions (warps) are scheduled every two clock cycles simultaneously
- Fast single-, double-, and half-precision
- High Bandwidth Memory 2 (HBM2) at 732 GB/s
- NVLink between multiple GPUs (20 GB/s in each direction)
- Unified virtual memory and paging support

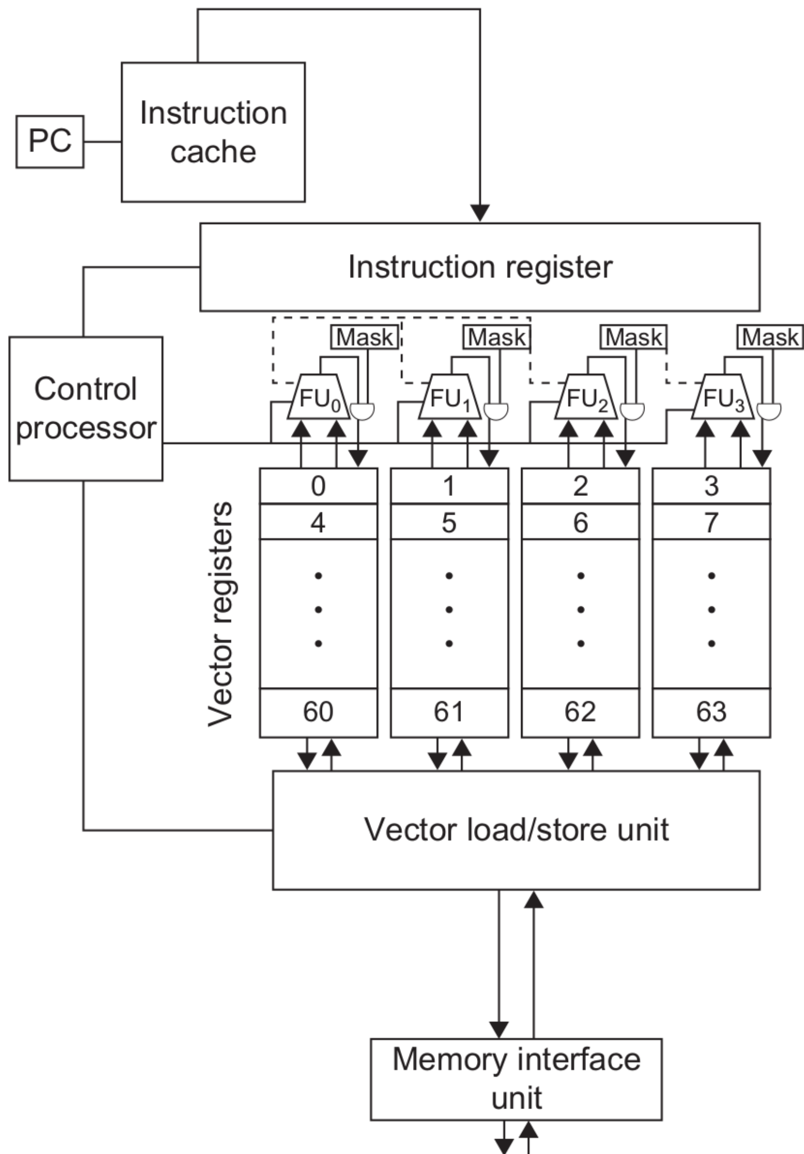
NVIDIA's Pascal SIMD Processor

GPUs vs. Vector Architectures

- Ομοιότητες με vector αρχιτεκτονικές
  - Δουλεύει επίσης καλά για data-level parallel προβλήματα
  - Scatter-gather transfers
  - Mask registers
  - Large register files

- Διαφορές με vector αρχιτεκτονικές
  - Δεν υπάρχει scalar επεξεργαστής
  - Χρησιμοποιεί multithreading για να κρύψει την καθυστέρηση στην πρόσβαση της μνήμης
  - Έχει πολλαπλά functional units, σε αντίθεση με τα λίγα deeply pipelined functional units που έχουν οι vector αρχιτεκτονικές

GPUs vs. Vector Architectures



Vector Architecture

PC
PC
PC
PC
Instruction cache
SIMD thread scheduler
Dispatch unit
Instruction register
Mask
FU$_0$
FU$_1$
FU$_2$
FU$_3$
Registers
0
1
1023
SIMD load/store unit
Address coalescing unit
Memory interface unit

GPU Architecture

SIMD Extensions vs. GPUs

- Οι GPUs έχουν περισσότερα SIMD lanes
- Οι GPUs υποστηρίζουν στο υλικό περισσότερα threads
- Και οι δύο έχουν 2:1 αναλογία μεταξύ double- and single-precision performance
- Και οι δύο έχουν 64-bit διευθύνσεις, αλλά οι GPUs έχουν μικρότερη μνήμη
- Τα SIMD extensions δεν υποστηρίζουν scatter-gather εντολές

Προσεγγίσεις Αύξησης Απόδοσης

- **Παραλληλισμός σε επίπεδο εντολής (Instruction Level Parallelism – ILP)**
  - Εξαρτάται από τις πραγματικές εξαρτήσεις δεδομένων που υφίστανται ανάμεσα στις εντολές.

- **Παραλληλισμός σε επίπεδο νήματος (Thread-Level Parallelism – TLP)**
  - Αναπαρίσταται ρητά από τον προγραμματιστή, χρησιμοποιώντας πολλαπλά νήματα εκτέλεσης τα οποία είναι εκ κατασκευής παράλληλα.

- **Παραλληλισμός σε επίπεδο δεδομένων (Data-Level Parallelism – DLP)**
  - Αναπαρίσταται ρητά από τον προγραμματιστή ή δημιουργείται αυτόματα από τον μεταγλωττιστή.
  - Οι ίδιες εντολές επεξεργάζονται πολλαπλά δεδομένα ταυτόχρονα