Περιήγηση Στην Αρχιτεκτονική:

Από Τον Intel 4004 Στον Intel Pentium 4 Σε Δύο (Περίπου) Ώρες.

Στέφανος Καξίρας

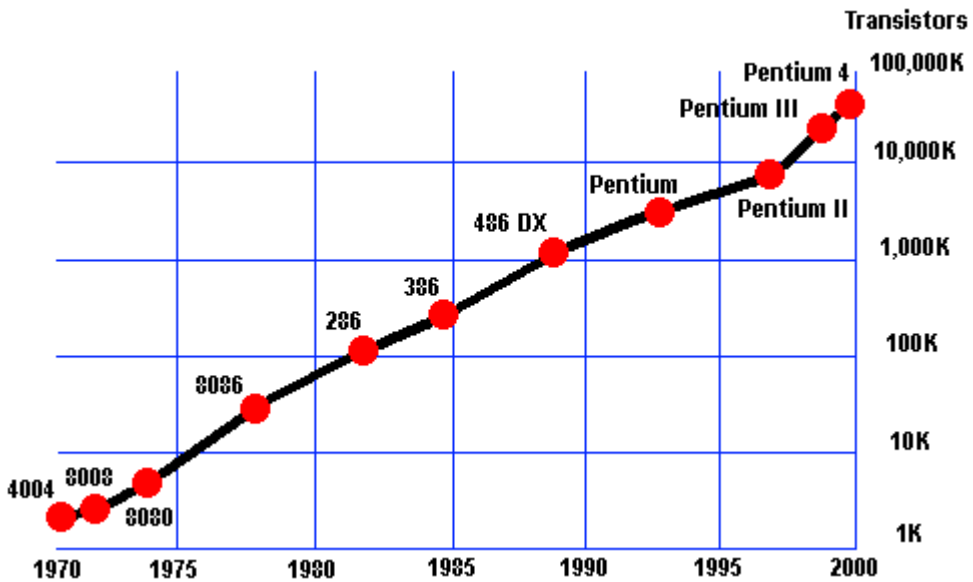
{ kaxiras@cs.wisc.edu, kaxiras@ee.upatras.gr }

Περιεχόμενα

- Αρχιτεκτονική και ο νόμος του Moore
- Σετ εντολών
  - Ο πόλεμος των σετ εντολών: RISC εναντίον CISC
- Από το μικροκώδικα στο pipelining
  - 80386 -> 80486
- Σπάζοντας το φράγμα του 1 IPC: superscalar
  - 80486 -> Pentium
- Το επόμενο βήμα: ΕΕΣ (OOO)
  - Pentium -> Pentium Pro/II/III, Pentium 4
- Μνήμη
  - Caches
- Κατανάλωση Ισχύος

Αρχιτεκτονική και ο Νόμος του Moore

- Νόμος του Moore
  - Ο αριθμός των τρανζίστορ στα ολοκληρωμένα κυκλώματα διπλασιάζεται κάθε ~ 2 χρόνια
  - Intel 4004 (1971): 2300, Pentium 4 (2001): 55.000.000



Αρχιτεκτονική και ο Νόμος του Moore

- Στην πραγματικότητα η απόδοση (ταχύτητα επεξεργασίας για συγκεκριμένα προγράμματα) των επεξεργαστών διπλασιάζεται σε λιγότερο από 2 χρόνια
- Σε λιγότερο από 2 χρόνια θα κάνουμε τόση πρόοδο όση κάναμε στα προηγούμενα 28 χρόνια ...

- Είναι αυτό αποτέλεσμα του νόμου του Moore?
  - Μικρότερα τρανζίστορ = πιο γρήγορα τρανζίστορ = πιο πολλά MHz = απόδοση ?
  - ΟΧΙ

Αρχιτεκτονική και ο Νόμος του Moore

- Η αρχιτεκτονική είναι αυτή που μεταφράζει τα κέρδη στην πυκνότητα και ταχύτητα των τρανζίστορ σε κέρδη στην απόδοση του επεξεργαστή.
- Παράδειγμα 1: Το χάσμα της μνήμης (memory gap/wall)
  - Την εποχή του Intel 8086 η μνήμη DRAM ήταν πιο γρήγορη από τον επεξεργαστή. Ένα LOAD πιο γρήγορο από ένα ADD.
  - Σήμερα ένα LOAD ~ 70 κύκλους του επεξεργαστή, ίσως ~ 140 εντολές (ADD) !
  - Αλλά η αναλογία των LOAD στα προγράμματα παραμένει η ίδια …
  - Λύση ? Αρχιτεκτονική:
    - Cache memories, “κρυφές” “γρήγορες” μνήμες
    - κάνουν τη DRAM να φαίνεται γρήγορη ...

Αρχιτεκτονική και ο Νόμος του Moore

- Παράδειγμα 2: παραλληλισμός στην εκτέλεση εντολών
  - Την εποχή του Intel 8086 μια εντολή χρειαζόταν πολλούς κύκλους για να εκτελεστεί
  - Σήμερα ένας Pentium 4 μπορεί να εκτελέσει ταυτόχρονα στον ίδιο κύκλο 6 εντολές
  - Εάν δεν μπορούσαμε να εκτελέσουμε πολλές εντολές κάθε κύκλο η αύξηση της απόδοσης θα υπολείπονταν της αύξησης συχνότητας (που έτσι και αλλιώς είναι μικρότερη από 2x/2,5χρόνια).
- Αυτά τα δύο παραδείγματα αποτελούν την ουσία της σύγχρονής αρχιτεκτονικής

Σετ εντολών

- Η αρχιτεκτονική του σετ εντολών (ISA) ενός επεξεργαστή είναι το interface με τον έξω κόσμο.
- Το σετ εντολών καθώς, η διαχείριση μνήμης (Εικονική μνήμη, virtual memory) και ο χειρισμός των διακοπών (interrupts) ορίζουν την συμπεριφορά του επεξεργαστή.
- Έχει σημασία το σετ εντολών για το πόσο εύκολο είναι αρχιτεκτονικά να αποσπάσουμε την μέγιστη απόδοση από έναν επεξεργαστή ?

Σετ εντολών

- Τη δεκαετία του 80 οι μικροεπεξεργαστές με αρχηγό την Intel αλλά και άλλους (π.χ. Moto. 68000, Nat. Semi. 32032) ακολουθούσαν τους κατασκευαστές των mini (IBM, DEC VAX-11) και ολοένα παρουσίαζαν πιο πολύπλοκα σετ εντολών
- Οι εντολές γινόταν όλο και πιο σύνθετες, πλησιάζοντας τις γλώσσες υψηλού επιπέδου (semantic gap, VAX 11)
- Η αποκορύφωση ήταν ο Intel 432, ένας φοβερά πολύπλοκος επεξεργαστής και μια φοβερή εμπορική αποτυχία της Intel.

Σετ εντολών

- Επικρατούσα αρχιτεκτονική: αρχιτεκτονική με μικροκώδικα:
  - ένα μικροπρόγραμμα που έτρεχε συνεχώς σε ένα υποτυπώδη κρυφό επεξεργαστή διάβαζε τις (μακρο)εντολές του προγράμματος μια-μια και ενορχήστρωνε την εκτέλεσή τους κατευθείαν πάνω στο υλικό χρησιμοποιώντας όμως πολλούς κύκλους για κάθε μακροεντολή
- Δεδομένου του περιορισμένου αριθμού τρανζίστορ
  - Αύξηση της πολυπλοκότητας του σετ εντολών με μεγαλύτερο μικροκώδικα
- Δυνατότερες εντολές = καλύτερη απόδοση ?
- CISC

Σετ εντολών

- RISC: Αντίδραση στην αύξηση της πολυπλοκότητας ήρθε από το Stanford (MIPS) και το Berkeley (RISC-I, RISC-II, SUN SPARC)
- Δεδομένου του περιορισμένου αριθμού τρανζίστορ
  - Αύξηση της απόδοσης με απλούστερο σετ εντολών (στα βήματα των CRAY και CDC 6600)
- Μικροεντολές -> σετ εντολών

RISC εναντίων CISC

	RISC	CISC	Διαφορά σε
Εντολές	32 bit	Μεταβλητού μήκους	Decode
	Ομοιόμορφες	Πολλές μορφές	Decode
	Απλές	Πολύπλοκες	Execute
Καταχωρητές	Πολλοί	Λίγοι	Pipelining
Addressing Modes	Λίγα	Πολλά	Decode
	Απλά	Πολύπλοκα	Execute
	Χρήσιμα σε compilers	Αχρησιμοποίητα από compilers	
Εκτέλεση	Load-store	Memory-memory	Pipelining

RISC εναντίων CISC

- ~~Δεδομένου του περιορισμένου αριθμού τρανζίστορ~~
- και με πάρα πολλούς πόρους διαθέσιμους...

	IA-32, x86(CISC)	Εσωτερική μετάφραση σε RISC μ-ops P-PRO/II/III/4
Εντολές	Μεταβλητού μήκους	Σταθερού μήκους 118bit
	Πολλές μορφές	-
	Πολύπλοκες	Απλές
Καταχωρητές	Λίγοι (αρχιτεκτονικοί)	Πάρα πολλοί εσωτερικοί (physical registers)
Addressing Modes	Πολλά	Μετάφραση σε πράξεις με διευθύνσεις
	Πολύπλοκα	
	Αχρησιμοποίητα από compilers	
Εκτέλεση	Memory-memory	Load-store εσωτερικα

Η κατάληξη

- Παρόλο που το RISC ήταν πολύ επιτυχημένο την εποχή που παρουσιάστηκε (2,6x RISC advantage), το CISC κατόρθωσε να επιζήσει και να επικρατήσει
- Το CISC μπόρεσε να εκμεταλλευτεί όλες τις καινοτομίες του RISC, κυρίως OOO, caches
- Η αρχιτεκτονική όπως εκφράζεται σήμερα αναφέρεται στο εσωτερικό RISC πυρήνα των επεξεργαστών (είτε είναι Pentium 4 είτε οτιδήποτε άλλο)
- Το "εξωτερικό" σετ εντολών δεν καθορίζει πια την "εσωτερική" αρχιτεκτονική ενός επεξεργαστή
  - TRANSMETA CRUSOE (Pentium compatible): μεταφράζει το IA-32(x86) με λογισμικό στο δικό του σετ εντολών (VLIW) και μετά το εκτελεί !

Περιεχόμενα

- Αρχιτεκτονική και ο νόμος του Moore
- Σετ εντολών
  - Ο πόλεμος των σετ εντολών: RISC εναντίον CISC
- Από το μικροκώδικα στο pipelining
  - 80386 -> 80486
- Σπάζοντας το φράγμα του 1 IPC: superscalar
  - 80486 -> Pentium
- Το επόμενο βήμα: OOO, Εκτέλεση Εκτός Σειράς
  - Pentium -> Pentium Pro/2/3, Pentium IV
- Μνήμη
  - Caches
- Κατανάλωση Ισχύος

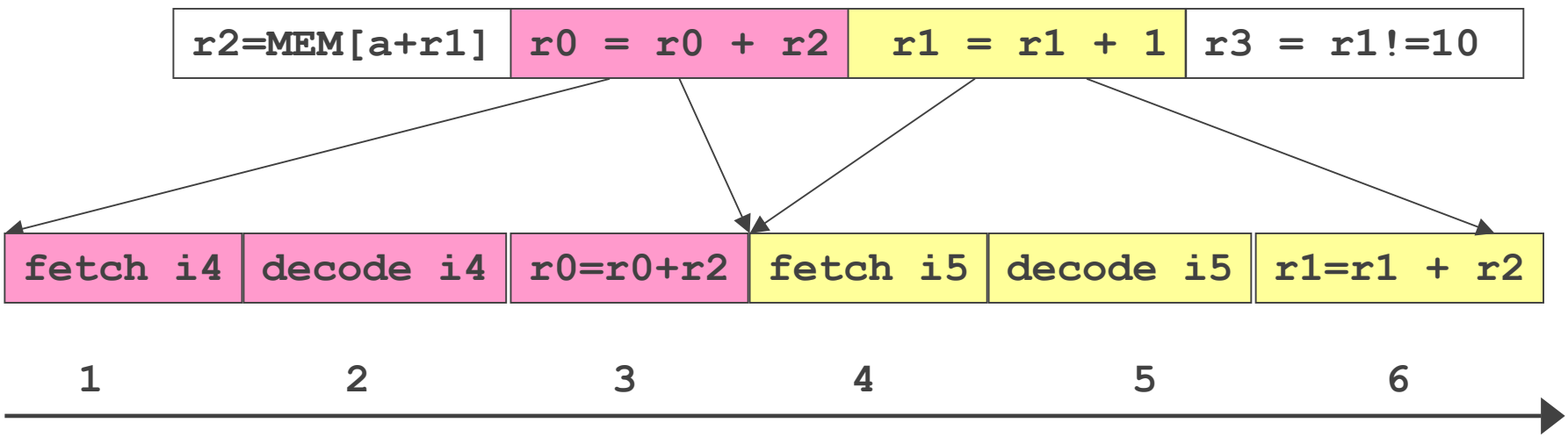
Από τον μικροπρογραμματισμό στο pipelining: 80386 -> 80486

- 80386 (1986), IA-32, x86
- 8 registers
- Αρχιτεκτονική μικροκώδικα
  - ένα μικροπρόγραμμα τρέχει συνεχώς σε ένα υποτυπώδη κρυφό επεξεργαστή, διαβάζει τις x86 εντολές του προγράμματος μια-μια και ενορχηστρώνει την εκτέλεσή τους κατευθείαν πάνω στο υλικό, χρησιμοποιώντας όμως πολλούς κύκλους για κάθε x86 εντολή
- π.χ. fetch - decode - execute

80386

```
sum=0
for i=0 to 10
    sum+=a[i]
```

```
i1       r1=0            /* index */
i2       r0=0            /* sum   */
i3 loop:r2=MEM[a+r1] /* a[i]  */
i4       r0=r0+r2
i5       r1=r1+1
i6       r3=(r1 != 10)
i7       if(r3==true) goto loop
```



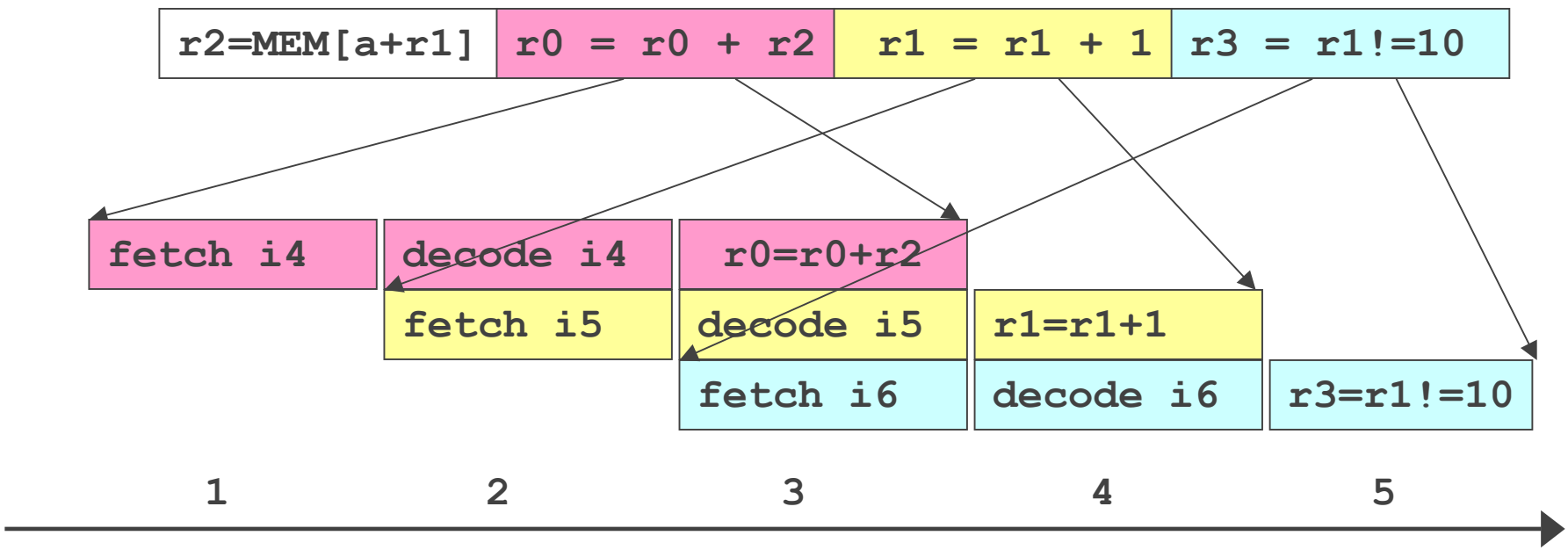
Από τον μικροπρογραμματισμό στο pipelining: 80386 -> 80486

- 80486 (1989) 1.000.000 τρανζίστορ, 2ο chip με > 1M τρανζίστορ (πρώτο με διαφορά μηνών το i860)
- Αρχιτεκτονική Pipeline/μικροκώδικα
- Pipeline: επικάλυψη των σταδίων εκτέλεσης των εντολών (π.χ. Fetch, decode, execute)
  - Σαν τις αλυσίδες παραγωγής αυτοκινήτων (π.χ., αμάξωμα, σαλόνι, μηχανή, βάψιμο)
- Ποιες εντολές εκτελούνται στο pipleline του 80486?
  - Απλές, RISC-like,
  - οι υπόλοιπες … με μικροκώδικα
- Pipeline παράδειγμα fetch - decode - execute (στον 80486 5 στάδια)

80486

```
sum=0
for i=0 to 10
    sum+=a[i]
```

```
i1       r1=0            /* index */
i2       r0=0            /* sum   */
i3 loop:r2=MEM[a+r1] /* a[i]  */
i4       r0=r0+r2
i5       r1=r1+1
i6       r3=(r1 != 10)
i7       if(r3==true) goto loop
```



Pipelining

- Απαράβατος Κανόνας: διατηρούνται τα Sequential Semantics, εντολές εκτελούνται με τη σειρά που εμφανίζονται στο πρόγραμμα
  - Εντολές χωρίζονται σε δύο κατηγορίες
    - In-progress: αποτελέσματά τους αόρατα στην εξωτερική κατάσταση του επεξεργαστή
    - Committed: τα αποτελέσματά τους ορατά στη κατάσταση του επεξεργαστή
- Το pipelining βελτιώνει μόνο το Ρυθμό (throughput) ΟΧΙ την καθυστέρηση (latency)
  - η κάθε εντολή είναι όσο αργή ήταν και πριν (π.χ. 3κ.)
  - 1000 εντολές 1002 κύκλους (χωρίς pipelining 3000 κύκλους). Στο ΟΡΙΟ 1 εντολή ανά κύκλο.
- Το pipeline πρέπει να είναι ΠΑΝΤΑ γεμάτο
  - Βλέπε branch prediction

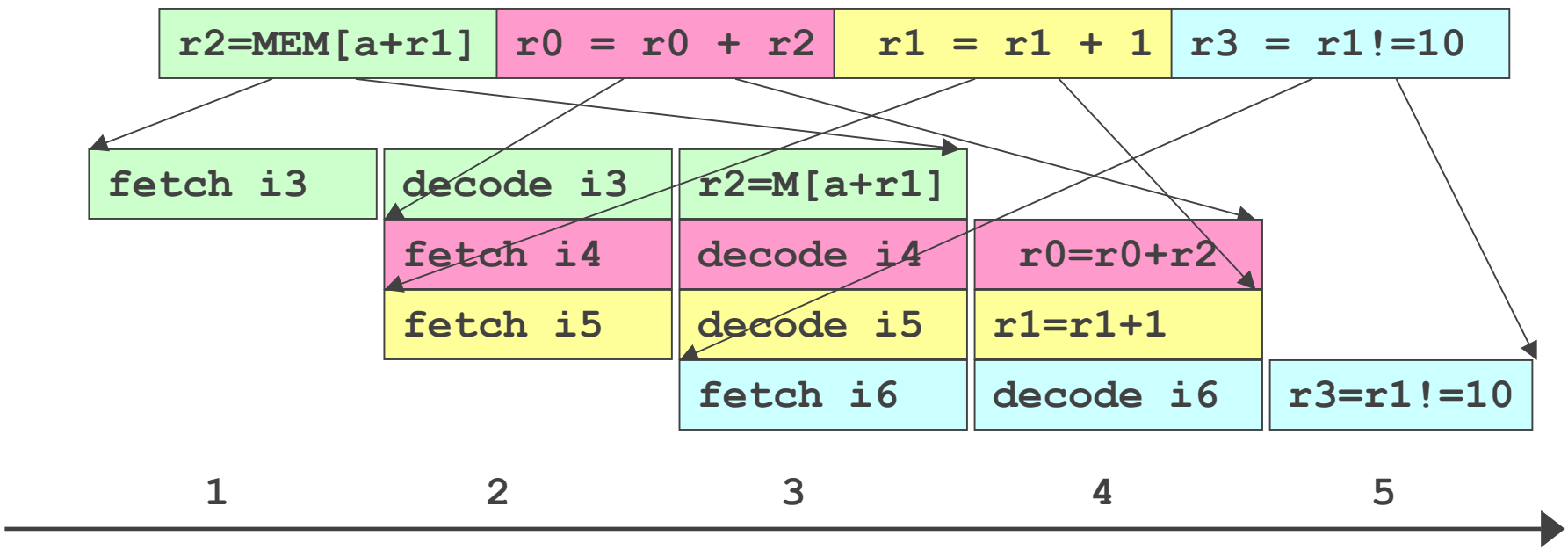
Superscalar: 80486 -> Pentium

- Pentium 1993, 3.100.000 τρανζίστορ με 16KB cache
- Εκτέλεση σε σειρά, μέχρι 2 εντολές τον κύκλο
- Ορισμός του superscalar: διαδοχικές εντολές παράλληλα
- Δύο pipelines, 5-στάδια η κάθε μία
- Ασύμμετρα pipelines: το ένα πιο "ικανό" από το άλλο
- Εύκολες, ανεξάρτητες διαδοχικές εντολές παράλληλα, διαφορετικά μία-μία στο κύριο pipeline
- Δύσκολες εντολές με μικροπρόγραμμα
- π.χ. Dual Superscalar Pipeline : fetch - decode - execute

Pentium

```
sum=0
for i=0 to 10
    sum+=a[i]
```

```
i1       r1=0            /* index */
i2       r0=0            /* sum   */
i3 loop: r2=MEM[a+r1]    /* a[i]  */
i4       r0=r0+r2
i5       r1=r1+1
i6       r3=(r1 != 10)
i7       if(r3==true) goto loop
```



Superscalar: 80486 -> Pentium

- Τι υλικό χρειάζεται για superscalar ?
- Στο decode πρέπει να αποφασίσουμε αν οι εντολές μπορούν να εκτελεστούν παράλληλα
  - Ανεξαρτησία εντολών: Εξάρτηση των εντολών εκδηλώνεται μέσα από την εγγραφή/ανάγνωση καταχωρητών:
    - RAW, WAR, WAW hazards (κίνδυνοι)
    - Μια εντολή χρειάζεται το αποτέλεσμα μιας άλλης
  - resource hazards: αν οι εντολές χρειάζονται τους ίδιους πόρους για την εκτέλεσή τους
- "Bypass networks" για τα pipelines και μεταξύ των pipelines
    - Παράκαμψη του πίνακα καταχωρητών (RF) για τη μεταφορά αποτελεσμάτων από εντολή σε εντολή
    - γράψιμο, ανάγνωση καταχωρητών πολύ αργό

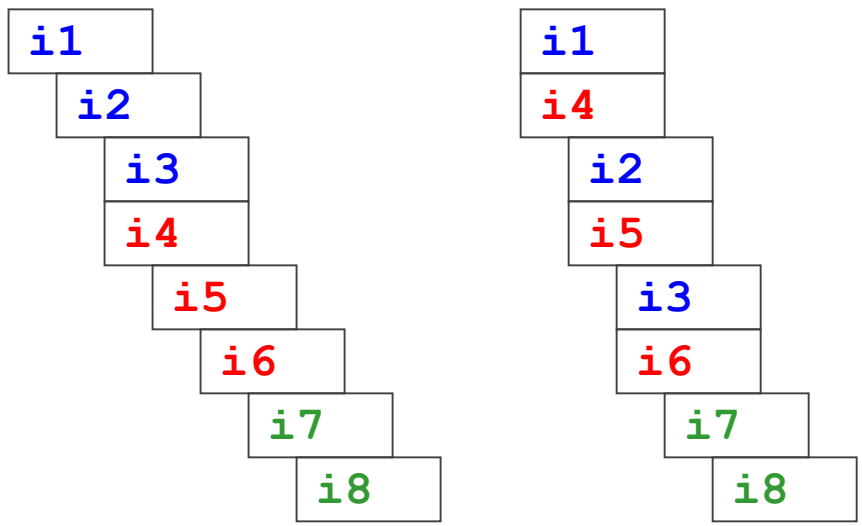
OOO: Pentium->Pentium Pro/II/III

- Pentium Pro (1995) 5,5M τρανζίστορ (2x8KB L1)
- Pentium II (1997) 7,5M - 27,4M (2x16KB L1, 256KB L2)
- Pentium III (1999) 9,5M - 44M (2x16KB L1, 512KB L2)
- Ο πρώτος ΟΟΟ επεξεργαστής. Dynamic Execution (Intel), Out-of-Order, Εκτέλεση εκτός σειράς προγράμματος
- ΟΟΟ ? Σαν το SUPERSCALAR αλλά οι εντολές που εκτελούνται παράλληλα ΔΕΝ χρειάζεται να είναι διαδοχικές
- Μετάφραση των x86 εντολών σε RISC μ-Ops
- 14 στάδια pipeline (12 σε σειρά 2 εκτός σειράς)
- μέχρι 5 μ-Ops παράλληλα σε 5 FUs:
  - 2 x ALU, LOAD, STORE, FPU

OOO: Pentium->Pentium Pro/II/III

- Γιατί χρειάζεται το OOO? Δεν φτάνει μόνο το SUPERSCALAR?
- Ας πούμε ότι έχουμε 2 pipelines διαθέσιμα

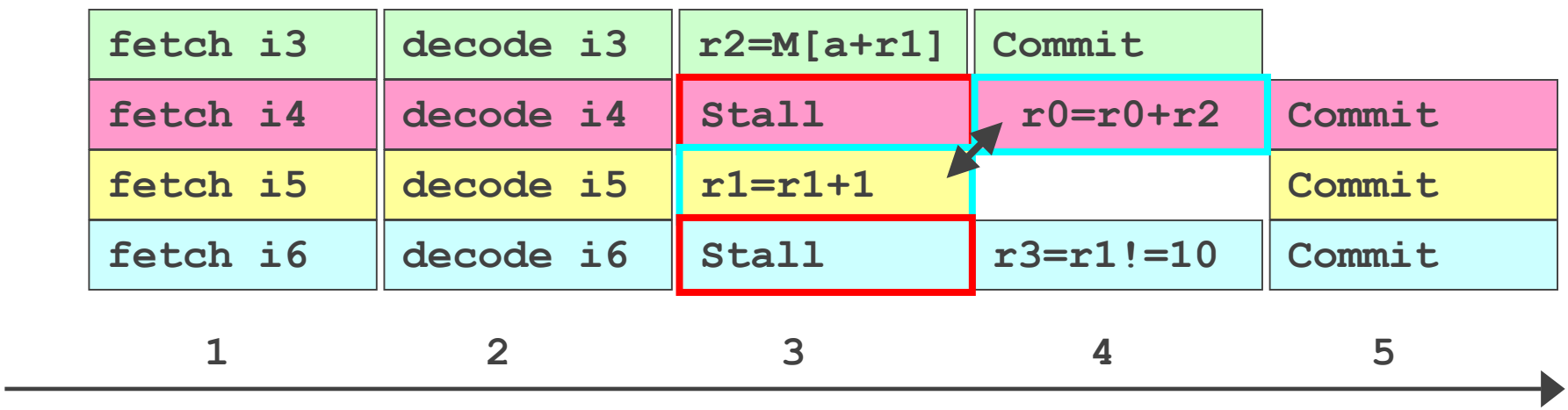
```
/* C =((A+1)/2)-((B+1)/2) */
i1 r1 = MEM[A]
i2 r2 = r1+1
i3 r3 = r1/2
i4 r4 = MEM[B]
i5 r5 = r4+1
i6 r6 = r5/2
i7 r7 = r3-r6
i8 MEM[C] = r7
```



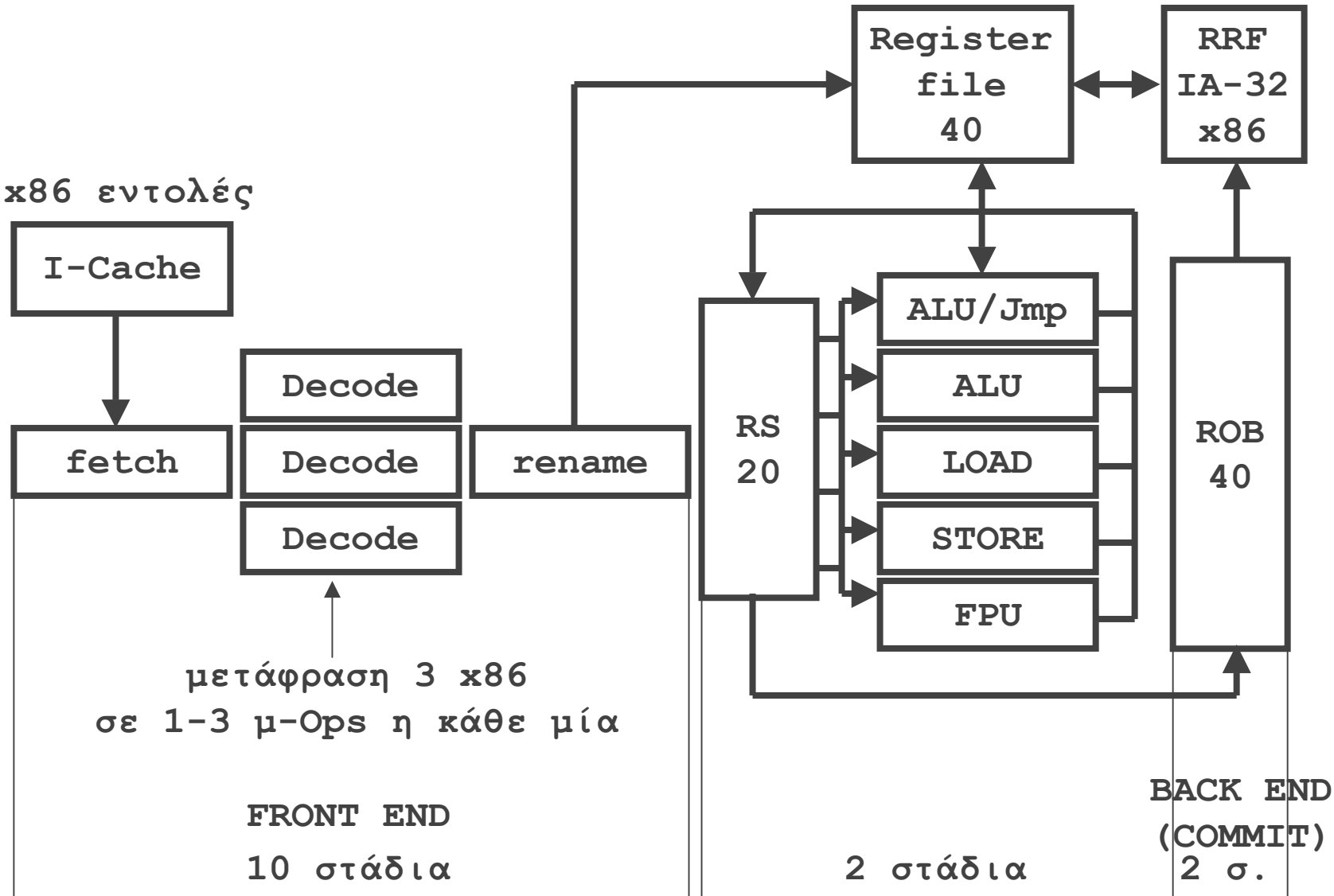
- Ο παραλληλισμός ΔΕΝ βρίσκεται σε διαδοχικές εντολές (που συνήθως εξαρτώνται η μια από την άλλη) ΑΛΛΑ ΠΙΟ ΜΑΚΡΥΑ

OOO: Pentium Pro/II/III

```
sum=0                          i1       r1=0           /* index */
for i=0 to 10                  i2       r0=0           /* sum   */
    sum+=a[i]                  i3 loop: r2=MEM[a+r1]   /* a[i]  */
                               i4       r0=r0+r2
                               i5       r1=r1+1
                               i6       r3=(r1 != 10)
                               i7       if(r3==true) goto loop
```

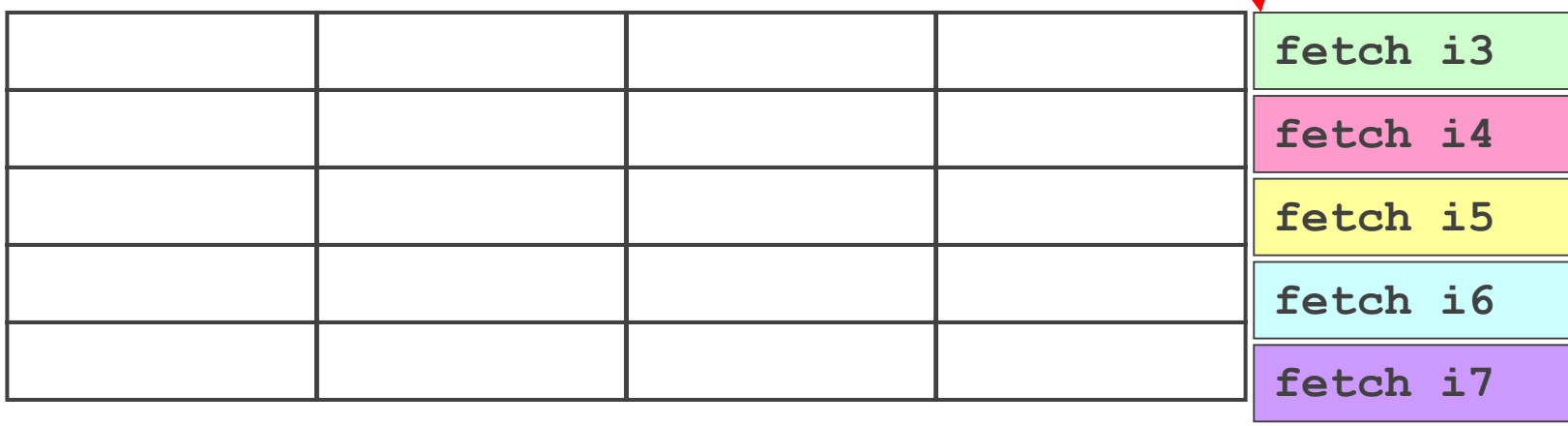


Pentium Pro/II/III block diagram (απλοποιημένο)



Το πρόβλημα με τα branch (JUMP)

fetch i3	decode i3	r2=M[a+r1]	Commit		
fetch i4	decode i4	Stall	r0=r0+r2	Commit	
fetch i5	decode i5	r1=r1+1		Commit	
fetch i6	decode i6	Stall	r3=r1!=10	Commit	
fetch i7	decode i7	Stall	Stall	if r3 loop	Commit



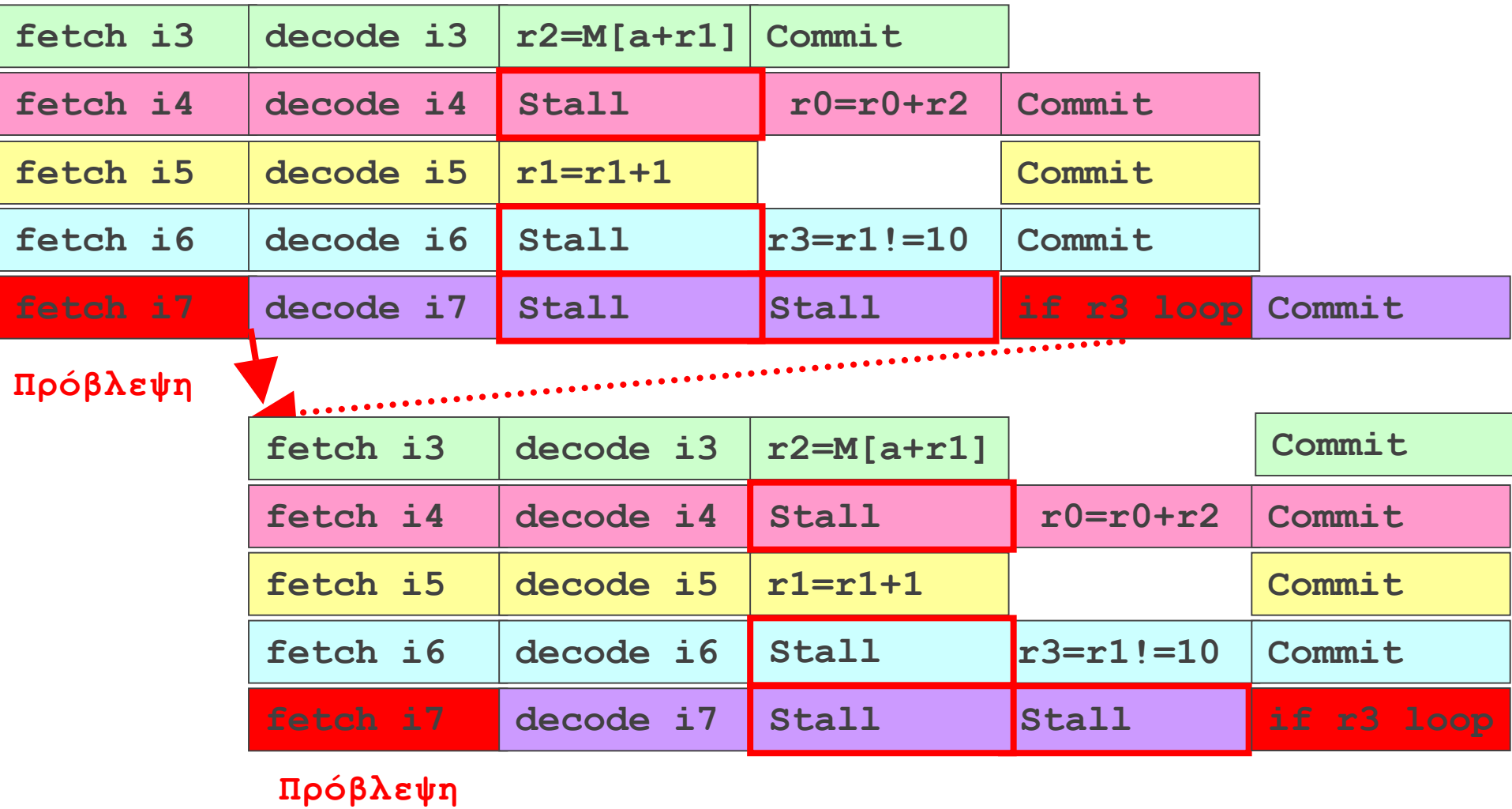
Branch prediction

- Pipeline Πάντα γεμάτο …
- Pentium Pro: 12 pipeline στάδια πριν από την εκτέλεση = 12 * 3 εντολές μετά από ένα branch (JUMP)
- Πού θα τις βρούμε όλες αυτές τις εντολές ?
  - Πρόβλεψη branch

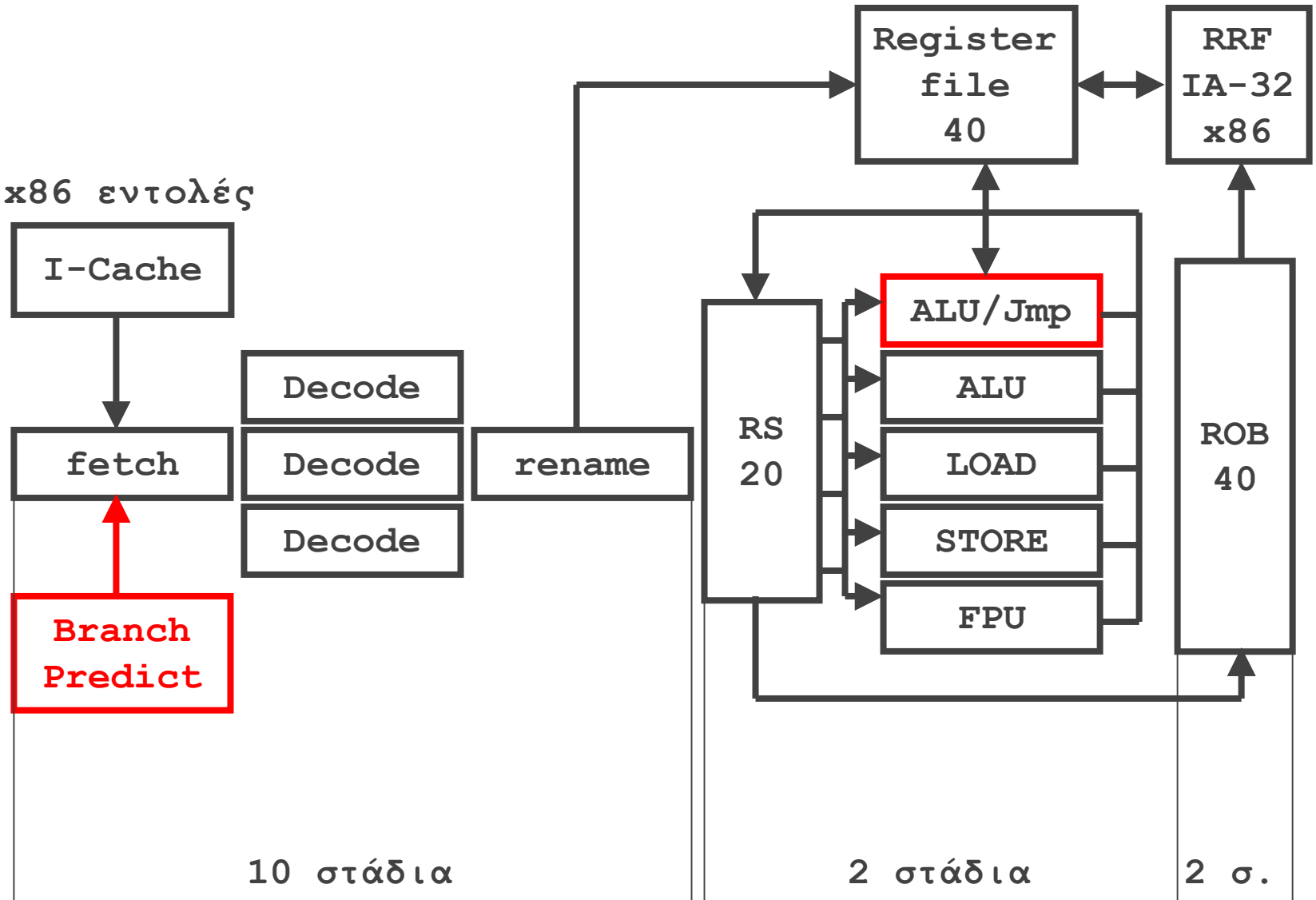
```
i1       r1=0            /* index */
i2       r0=0            /* sum   */
i3 loop:r2=MEM[a+r1] /* a[i]  */
i4       r0=r0+r2
i5       r1=r1+1
i6       r3=(r1 != 10)
i7       if(r3==true) goto loop
i8 continue: ...
```

  - Ποντάρουμε στο ότι το i7 μάλλον θα πάει στο i3 και όχι στο i8

Branch prediction



Pentium Pro/II/III block diagram, branch prediction

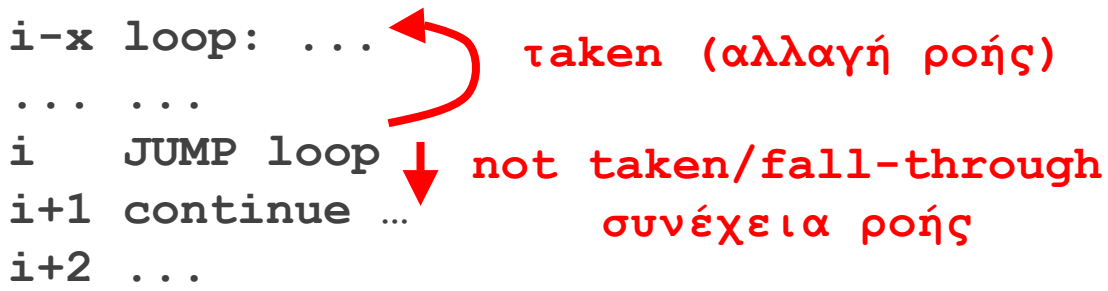


Speculative execution

- Η πρόβλεψη branch (JUMP) οδηγεί σε ΥΠΟΘΕΤΙΚΗ ΕΚΤΕΛΕΣΗ ΕΝΤΟΛΩΝ (Speculative Execution)
  - υποθέτουμε ότι θα εκτελεστούν …
- Αν όμως κάνουμε λάθος ?
  - Θα πρέπει τα αποτελέσματα της υποθετικής εκτέλεσης να είναι αναστρέψιμα
  - Οι βασικοί μηχανισμοί υπάρχουν ήδη για το OOO
  - Το ROB κρατάει τις υποθετικές εντολές ώσπου να επαληθευτεί η ροή του προγράμματος
  - ROB (40) > RS (20)

Branch prediction

- Branch (JUMP) ορολογία:



- Πως γίνεται το branch prediction?
  - Στατικά (π.χ. branch προς τα πίσω είναι “taken”)
  - Με τη βοήθεια του compiler
  - Δυναμικά (κατά την εκτέλεση του προγράμματος)

Branch prediction

- Δυναμικό branch prediction
  - Απλοϊκό: για κάθε branch κάνε ότι έκανε την προηγούμενη φορά
    - loop: 10 φορές σωστό, λάθος, λάθος, 9 φορές σωστό, λάθος, λάθος …
  - για κάθε branch κάνε ότι έτεινε να κάνει πιο πολύ τις *2-3* πιο πρόσφατες φορές που το είδαμε (Jim Smith, UW, βραβείο Mauchley-Eckert), Pentium branch prediction, 256 br.
    - loop: 10 φορές σωστό, λάθος, 10 φορές σωστό, λάθος...

Branch prediction

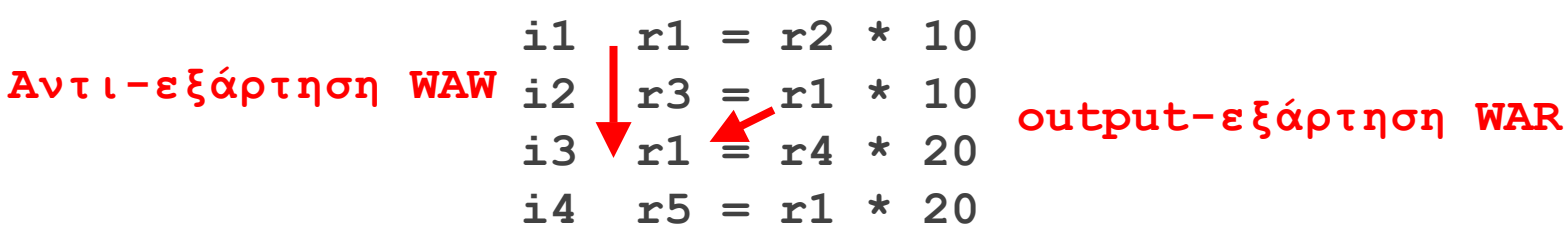
- Σύγχρονο branch prediction στον Pentium Pro/II/III
- Δύο επιπέδων, με ιστορία (two-level, history-based), Yeh & Patt, UM
- Η συμπεριφορά του branch εξαρτάται από το τι έγινε με τα άλλα branch πριν από αυτό
- Με άλλα λόγια από πιο δρόμο φτάσαμε σε αυτό το branch
  - Ιστορία: τι έγινε με τα *v* προηγούμενα branches
    - π.χ. taken,fall-through,taken,taken (1,0,1,1)
  - Πρόβλεψη: τι έτεινε να κάνει το branch (πιο πολύ) τις προηγούμενες 2-3 φορές που το είδαμε με την ίδια ιστορία
- Pentium Pro: 4 προηγούμενα branch ιστορία (πχ. 1011) , προβλέπει 512 συνδιασμούς ιστορίας/branch
- 95% ακρίβεια πρόβλεψης

Branch prediction

- Πολύ προσοδοφόρο πεδίο (πολλές δημοσιεύσεις)
- Γιατί ? 90-95% επιτυχία πρόβλεψης δεν είναι αρκετή …
- Π.χ.: καθημερινή πρόβλεψη σεισμών: σήμερα δεν θα γίνει σεισμός: 99% επιτυχία
  - Το θέμα είναι όμως πόσο “κοστίζει” η 1 (στις 100) λάθος πρόβλεψη
  - Στους επεξεργαστές το κόστος λάθος πρόβλεψης αυξάνει διαρκώς (Pentium 4, 20-στάδια pipeline, 6 εντολές παράλληλα …)
- Παραμένει ένα από τα δύσκολα προβλήματα που οδήγησε την Intel σε άλλες αρχιτεκτονικές : IA-64, Itanium που αποφεύγουν την πρόβλεψη

Register renaming

- Ο παραλληλισμός δεσμεύεται από τις αλληλεξαρτήσεις των εντολών
  - Αληθινές εξαρτήσεις: δεν μπορεί να εκτελεστεί μια εντολή πριν από τις εντολές που παράγουν τα ορίσματά της.
    - RAW (Read-After-Write) εξαρτήσεις.
  - Πλασματικές εξαρτήσεις: αποτροπή εκτέλεσης λόγω ανακύκλωσης των καταχωρητών
    - WAW (Write-After-Write) αντι-εξάρτηση
    - WAR (Write-After-Read) output εξάρτηση



Register renaming

- Για τις αληθινές εξαρτήσεις χρειάζεται να περάσουμε σε εξελιγμένα είδη πρόβλεψης για να τις υπερβούμε (Value Prediction, dataflow barrier). Θεωρητικά μιλώντας...
- Οι πλασματικές εξαρτήσεις μειώνουν τον παραλληλισμό στο επίπεδο εντολών (ILP) χωρίς να έχουν ουσιαστικό λόγο ύπαρξης. Απαλείφονται εύκολα αρκεί να είχαμε πάρα πολλούς καταχωρητές.
- Αλλά το Σετ Εντολών ποτέ δεν αλλάζει (συμβατότητα λογισμικού). (μόνο επεκτείνεται …)
- Λύση ? Μετονομασία Καταχωρητών (Register Renaming): Αντιστοίχισε τους καταχωρητές σε ένα πολύ μεγαλύτερο σετ, κατά την εκτέλεση του προγράμματος, αυτόματα και αόρατα στον εξωτερικό κόσμο.
- Εξαλείφει το πλεονέκτημα των RISC που πάντα υποστήριζαν πολλούς καταχωρητές για τους ίδιους λόγους

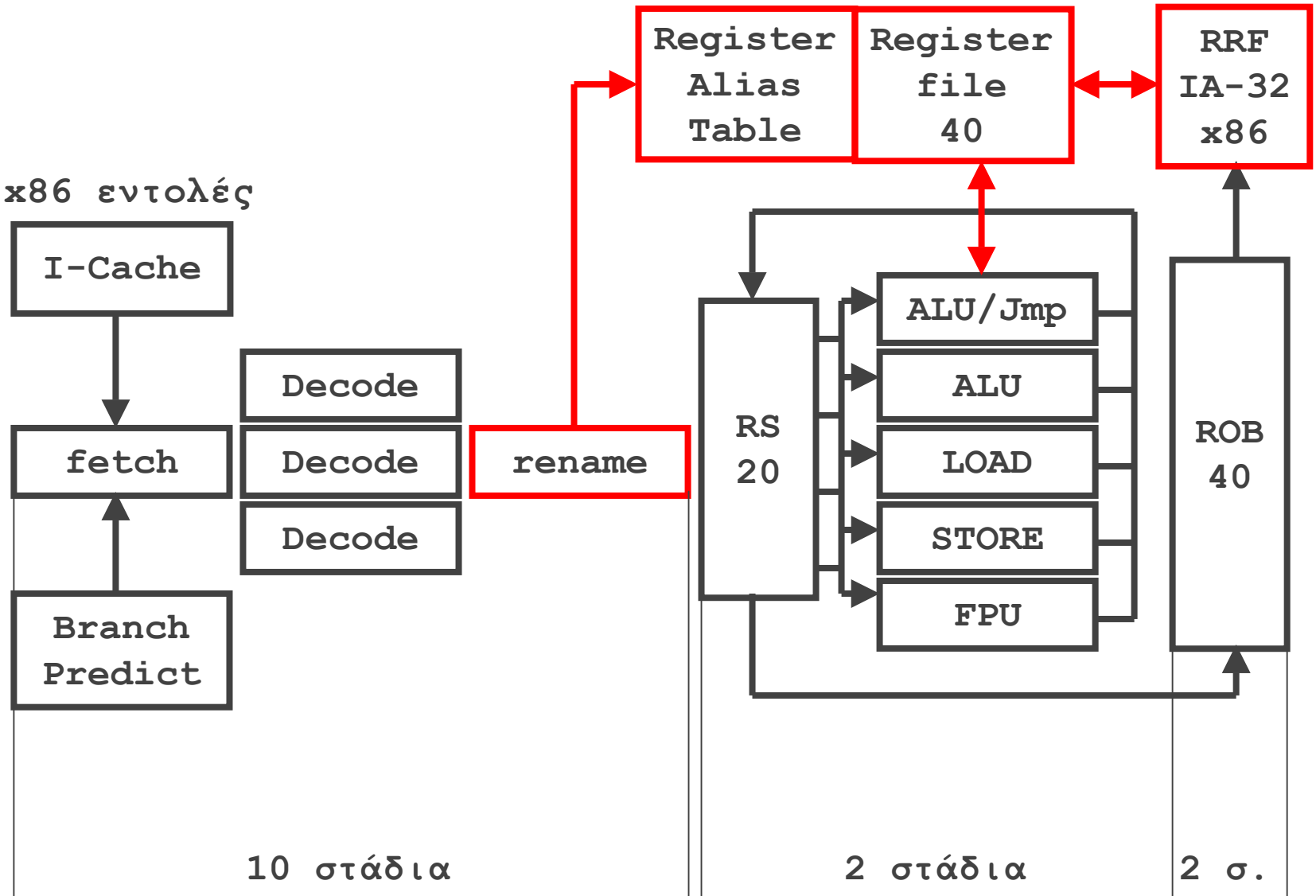
Register renaming

- Οι 8 "αρχιτεκτονικοί" καταχωρητές των x86 αντιστοιχίζονται σε 40 "φυσικούς" (physical) καταχωρητές
- Ένας πίνακας (RAT, Register Alias Table) κρατά τις αντιστοιχίες
- Κάθε φορά που γράφεται ένας αρχιτεκτονικός καταχωρητής αντιστοιχίζεται σε νέο φυσικό καταχωρητή

```
i1  r1 = r2 * 10        i1  ra = r2 * 10        i1  ra = r2 * 10
i2  r3 = r1 * 10        i2  r3 = ra * 10        i2  rb = ra * 10
i3  r1 = r4 * 20        i3  r1 = r4 * 20        i3  rc = r4 * 20
i4  r5 = r1 * 20        i4  r5 = r1 * 20        i4  r5 = rc * 20
```

- Με αυτό τον τρόπο εξαλείφουμε τις πλασματικές εξαρτήσεις (WAW, WAR) αλλά αφήνουμε ανέπαφες τις πραγματικές εξαρτήσεις (RAW)

Pentium Pro/II/III block diagram, register renaming



Σύνοψη: OOO

- Για να μεγιστοποιήσουμε τον παραλληλισμό στην εκτέλεση εντολών πρέπει να καταφύγουμε σε εκτέλεση εκτός σειράς προγράμματος (OOO).
- Το OOO υποβοηθείται από:
  - Πρόβλεψη ροής (Branch prediction) και υποθετική εκτέλεση (speculative execution)
    - επιτρέπει (1) να βρούμε αρκετές εντολές ώστε να κρατάμε όλα τα pipelines γεμάτα, (2) να εκτελούμε εντολές υποθετικά ώσπου να επαληθεύσουμε τη ροή του προγράμματος
  - Μετονομασία καταχωρητών (Register Renaming)
    - αφαιρεί τις πλασματικές εξαρτήσεις μεταξύ εντολών που προέρχονται από την ανακύκλωση καταχωρητών

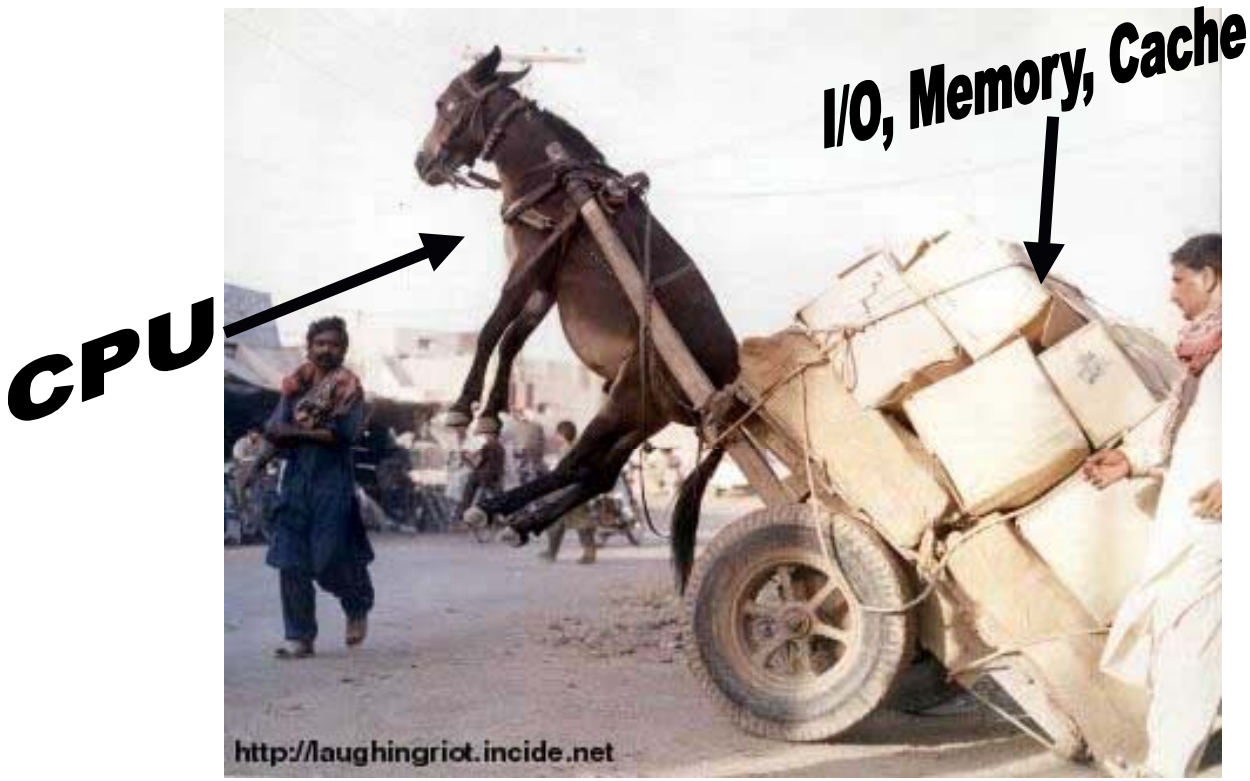
Περιεχόμενα

- Αρχιτεκτονική και ο νόμος του Moore
- Σετ εντολών
  - Ο πόλεμος των σετ εντολών: RISC εναντίον CISC
- Από το μικροκώδικα στο pipelining
  - 80386 -> 80486
- Σπάζοντας το φράγμα του 1 IPC: superscalar
  - 80486 -> Pentium
- Το επόμενο βήμα: OOO, Εκτέλεση Εκτός Σειράς
  - Pentium -> Pentium Pro/2/3, Pentium IV
- Μνήμη
  - Caches
- Κατανάλωση Ισχύος

Μνήμη

- Bob Colwell (αρχιτέκτονας του Pentium Pro) ISCA 2002 keynote address

An Unbalanced System



Μνήμη

- Το χάσμα της Μνήμης (Memory Wall, Memory Gap)
- Η ταχύτητα των επεξεργαστών αυξάνεται πολύ πιο πολύ από αυτή της μνήμης (DRAM)

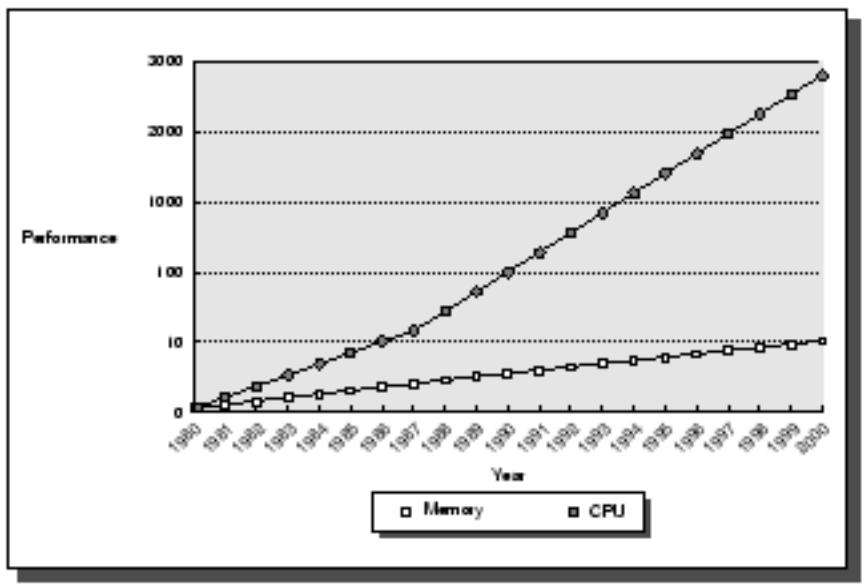


FIGURE: Starting with 1980 performance as a baseline, the performance of memory and CPUs are plotted over time.

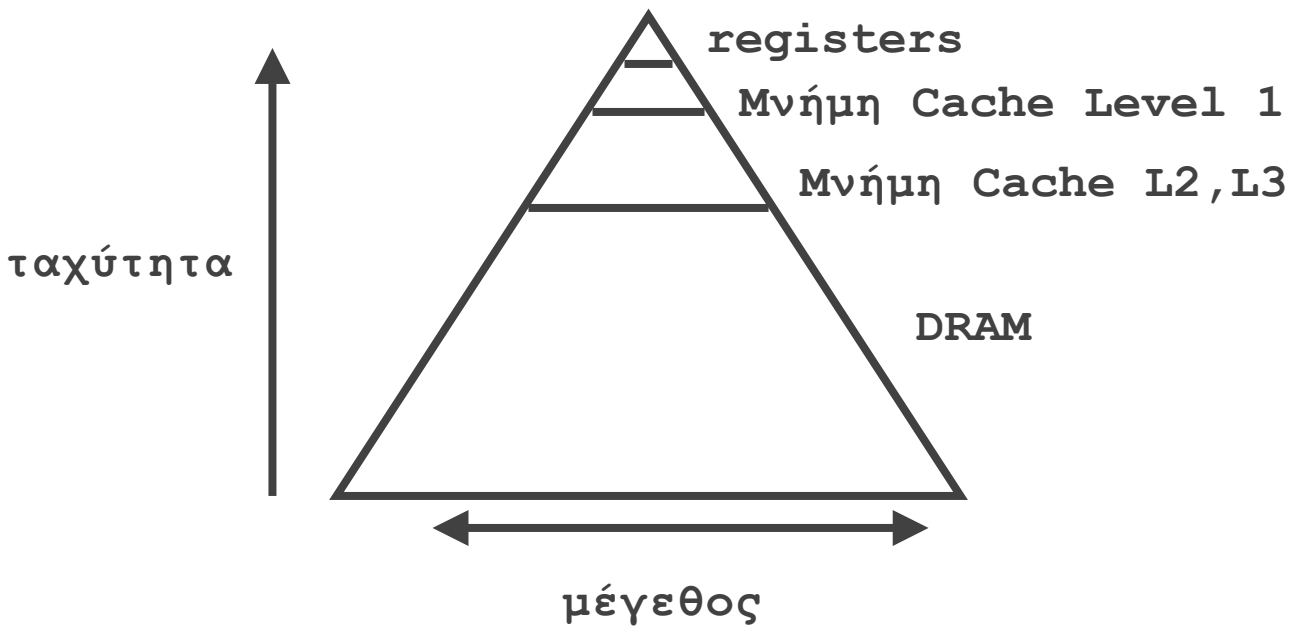
CPU 2x/2 χρόνια

DRAM 2x/10 χρόνια

- Το χάσμα αυξάνεται 50% το χρόνο …

Μνήμη

- Λύση ?
  - Ουσιαστικό χαρακτηριστικό της μνήμης: Η μνήμη μπορεί να είναι ΤΕΡΑΣΤΙΑ 'Η ΓΡΗΓΟΡΗ αλλά ΟΧΙ και τα δύο μαζί (VAST OR FAST)
  - => Ιεραρχία μνήμης:



Cache

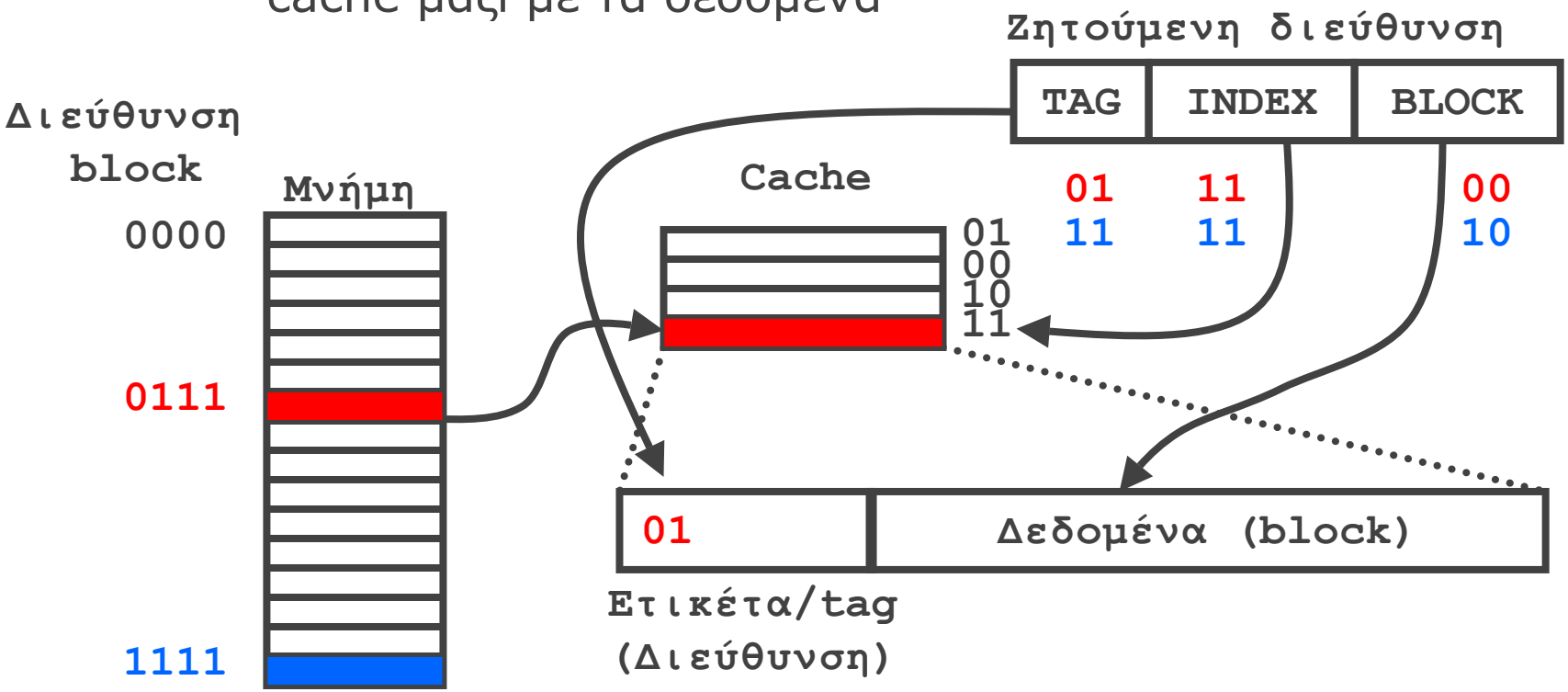
- Μνήμες Cache
  - SRAM, κοντά στον επεξεργαστή, αντικατοπτρίζουν τα πιο "χρήσιμα" περιεχόμενα της κύριας μνήμης
  - Κάνουν την κύρια μνήμη να "φαίνεται" πιο γρήγορη
  - Χαρακτηριστικά προσπέλασης μνήμης
    - Temporal Locality
      - Ότι προσπελάστηκε πρόσφατα είναι πιθανό να προσπελαστεί ξανά
    - Spatial Locality
      - Αν προσπελαστεί μια θέση μνήμης τότε είναι πιθανό ότι θα προσπελαστούν και οι κοντινές σε αυτή

Cache

- Λειτουργία
  - Σε κάθε προσπέλαση κοιτάμε αν τα περιεχόμενα της μνήμης στη ζητούμενη διεύθυνση "αντικατοπτρίζονται" στην cache.
  - Ναι: cache HIT
    - Hit rate: ποσοστό προσπελάσεων που πετυχαίνουν
  - Όχι: cache MISS
    - Θέση στη cache κενή ? Φέρε τα δεδομένα από τη μνήμη
    - Θέση στη cache πιασμένη από άλλα δεδομένα (άλλη διεύθυνση μνήμης) ? cache REPLACEMENT:
      - γράψε στη μνήμη τα παλιά δεδομένα (αν χρειάζεται) και φέρε τα νέα
    - miss rate: ποσοστό που αποτυχαίνει (100-hit rate)

Cache

- Πως βρίσκουμε τι υπάρχει στην cache και τι όχι ?
- Ζητούμενη διεύθυνση χωρίζεται σε 3 μέρη: block, index, tag
- Το index μας δίνει τη θέση στη cache που θα ψάξουμε
- Το tag (ετικέτα) συγκρίνεται με αυτό που φυλάσσεται στην cache μαζί με τα δεδομένα



Χαρακτηριστικά Cache

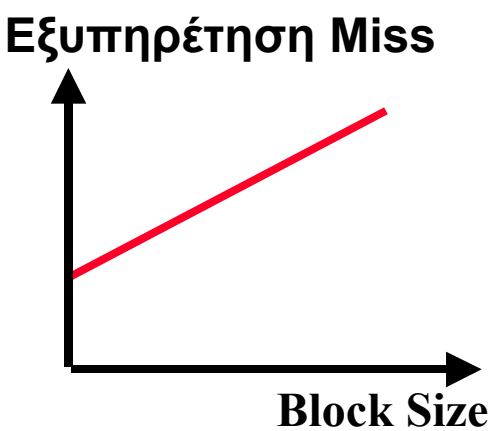
- Είδος δεδομένων
  - εντολές, instruction cache
  - δεδομένα, data cache
  - ενοποιημένη, unified cache
- Τοποθεσία
  - εσωτερική
  - εξωτερική
  - L1 (level 1) Επίπεδο Ιεραρχίας, L1 πιο κοντά στον επεξεργαστή
  - L2, L3

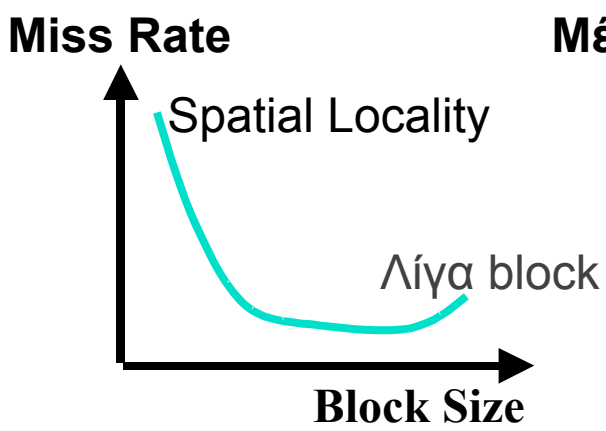
Χαρακτηριστικά Cache

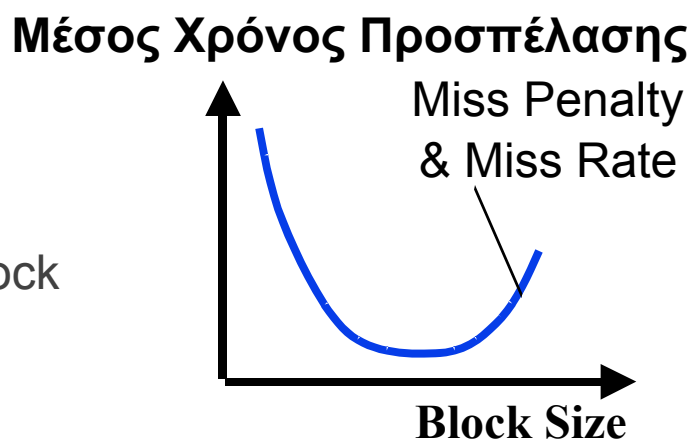
- Αρχιτεκτονική cache
  - block size (16-64 bytes)
  - direct-mapped
  - set-associative
  - write-through vs. write-back
  - blocking vs. non-blocking
  - prefetch

Αρχιτεκτονική Cache: μέγεθος block

- Η μνήμη cache μεταφέρει δεδομένα από και προς τη μνήμη σε block
  - Κόστος των Tag (block 1 byte, 66% της cache είναι tags)
  - Spatial locality, μεγαλύτερα block καλύτερα, αλλά ...
  - Βοηθά στη καλή αξιοποίηση του BUS (Burst mode)
    - Χρόνος μεταφοράς (αυξάνει με το μέγεθος του block)

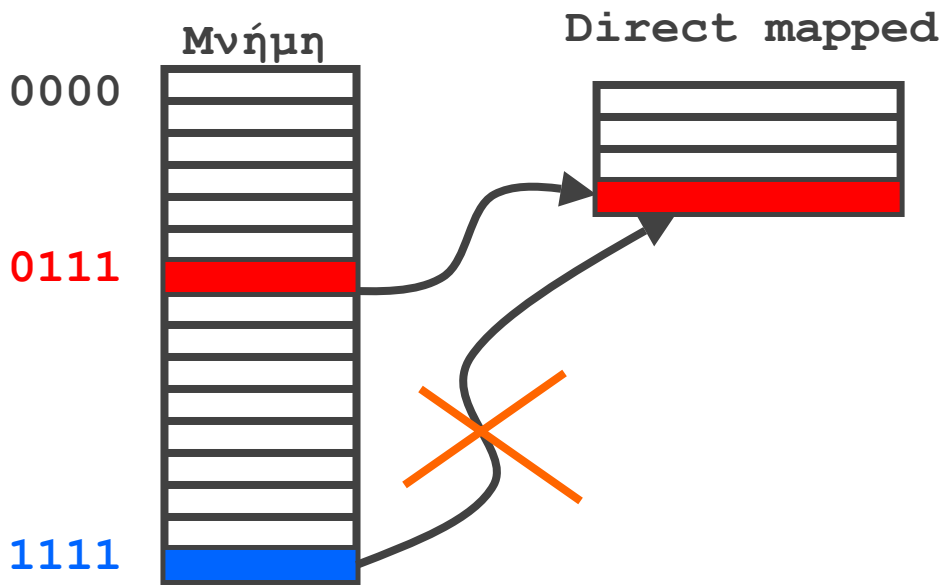






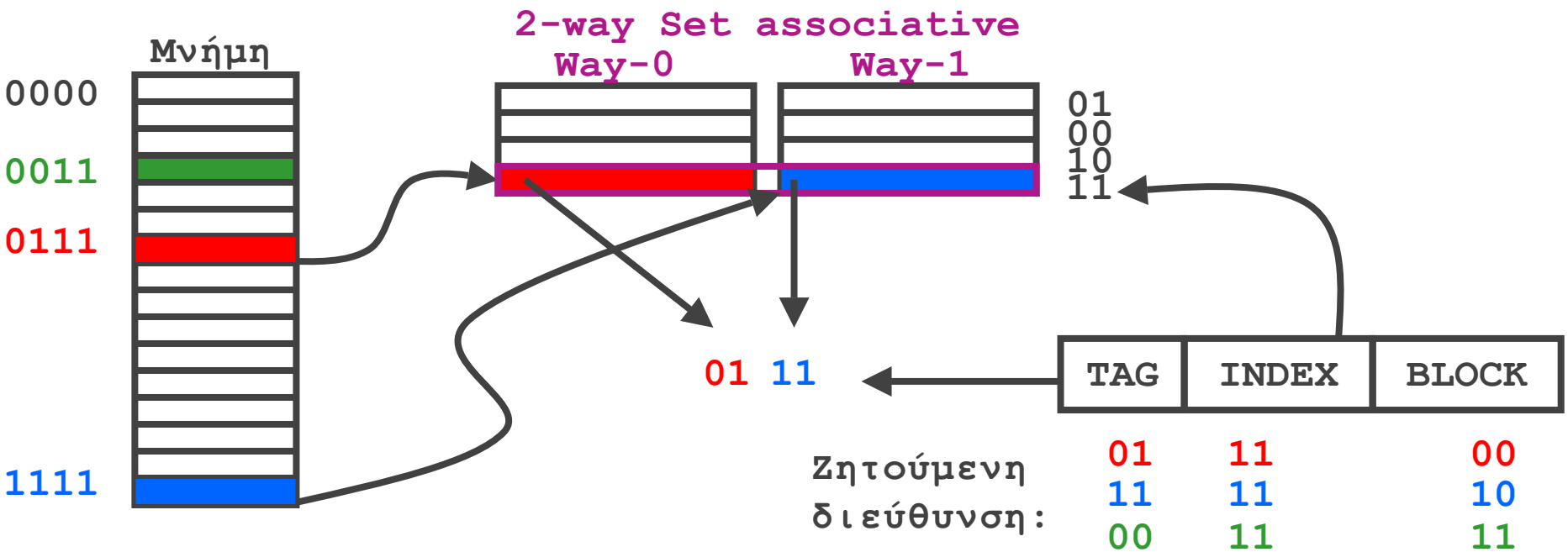
Αρχιτεκτονική Cache: direct-mapped

- Direct mapped
  - 1-1 αντιστοιχία block μνήμης σε θέση στην cache



Αρχιτεκτονική Cache: set-associative

- Set associative (n-way):
  - 1-n αντιστοιχία block μνήμης σε θέσεις στην cache
  - Το index διαλέγει ένα ολόκληρο set
  - Πρέπει να συγκρίνουμε με όλα τα tags στο set για να βρούμε αν κάτι είναι στην cache
  - Καλύτερο Hit Rate, πιο αργή στην προσπέλαση



Αρχιτεκτονική Cache: set-associative

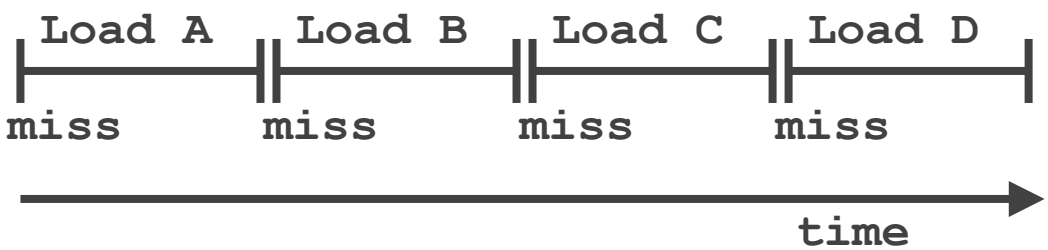
- Αντικατάσταση
  - Τυχαία (Random)
    - πολύ εύκολη υλοποίηση
    - αποδεκτή απόδοση
  - LRU (Pseudo-LRU): Least Recently Used, λιγότερο πρόσφατο
    - δύσκολη υλοποίηση
    - καλύτερη απόδοση
    - Pseudo-LRU: συμβιβασμός
  - FIFO: First-In First-Out
- 2 με 4 way η καλύτερη απόδοση
  - πιο πολλά way: πολύ μικρή αύξηση στην απόδοση με μεγάλο κόστος

Αρχιτεκτονική Cache: write-trough vs. write-back

- Write-through
  - Κάθε φορά που γράφουμε σε ένα block της cache στέλνουμε τα νέα δεδομένα και στη μνήμη
- Write-back
  - Νέα δεδομένα που γράφονται στην cache δεν στέλνονται στη μνήμη αμέσως αλλά μόνο αν αντικατασταθεί το block
  - Πολλαπλές εγγραφές στο block χρειάζονται μόνο μια μεταφορά στην μνήμη
  - Τα περιεχόμενα της μνήμης και της cache ΔΕΝ είναι ιδια Cache Consistency problem => I/O, multiprocessing κλπ.
    - Αντιμετωπίζεται σαν μέρος του Cache Coherency Protocol (MESI) για multiprocessing
  - Χρειάζεται ένα επιπλέον bit το “dirty bit” σε κάθε block που υποδηλώνει εάν το block έχει αλλάξει

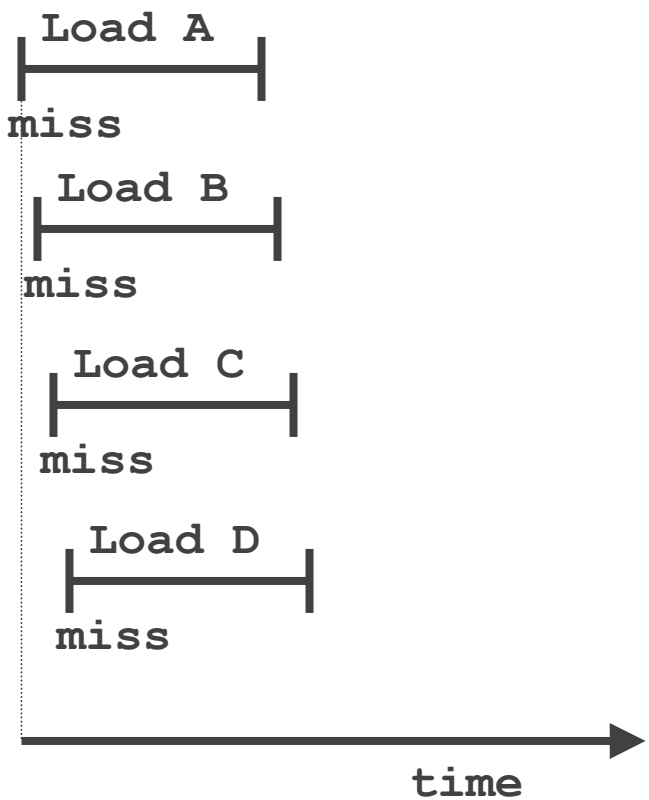
Αρχιτεκτονική Cache: blocking vs. non-blocking

- Γιά Superscalar και OOO χρειάζεται να εκτελούμε πολλά loads ταυτόχρονα (stores υπάρχει store buffer)
  - Οι σύγχρονες cache έχουν πολλαπλά ports και μπορούν να εξυπηρετήσουν πολλά Hits ταυτόχρονα
- Τι γίνεται όμως όταν έχουμε πολλαπλά miss ?
  - Εάν η cache μπορεί να εξυπηρετήσει μόνο ένα miss τη φορά (Blocking) τότε τα load εκτελούνται το ένα μετά το άλλο με μεγάλη καθυστέρηση



Αρχιτεκτονική Cache: blocking vs. non-blocking

- Λύση ? Non-blocking caches: εξυπηρετούν ταυτόχρονα πολλαπλά miss. PIPELINING για τα miss
- Το status κάθε miss ελέγχεται από ένα MSHR (miss-handling register)

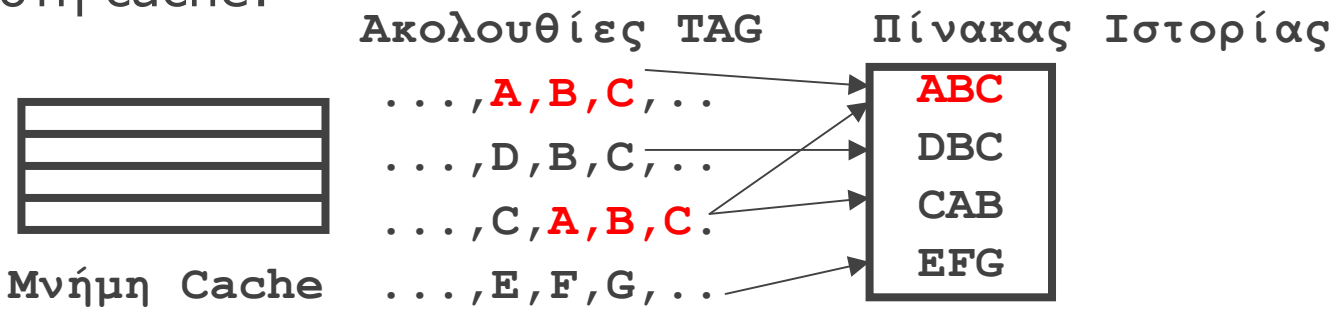


Αρχιτεκτονική Cache: prefetching

- Prefetching: μεταφορά block από την μνήμη στην cache πριν αυτά ζητηθούν από τον επεξεργαστή
  - Χρειάζονται περισσότερο bandwidth αλλά μπορεί να μειώσουν το latency
  - Οταν είναι λάθος μπορεί να κάνουν μεγάλη ζημιά: όχι μόνο καταναλώνουν bandwidth αλλά μπορεί να εκτοπίσουν χρήσιμα δεδομένα από την cache
- Πως γίνεται ?
  - Software, MIPS 10K
  - Hardware: πάρα πολλά σχήματα απλά … πολύ πολύπλοκα
    - sequential prefetch (block i, prefetch i+1)
    - stream buffers (ανακαλύπτουν ακολουθίες από block)
    - correlating prefetchers (πρόβλεψη με ιστορία)

Αρχιτεκτονική Cache: prefetching

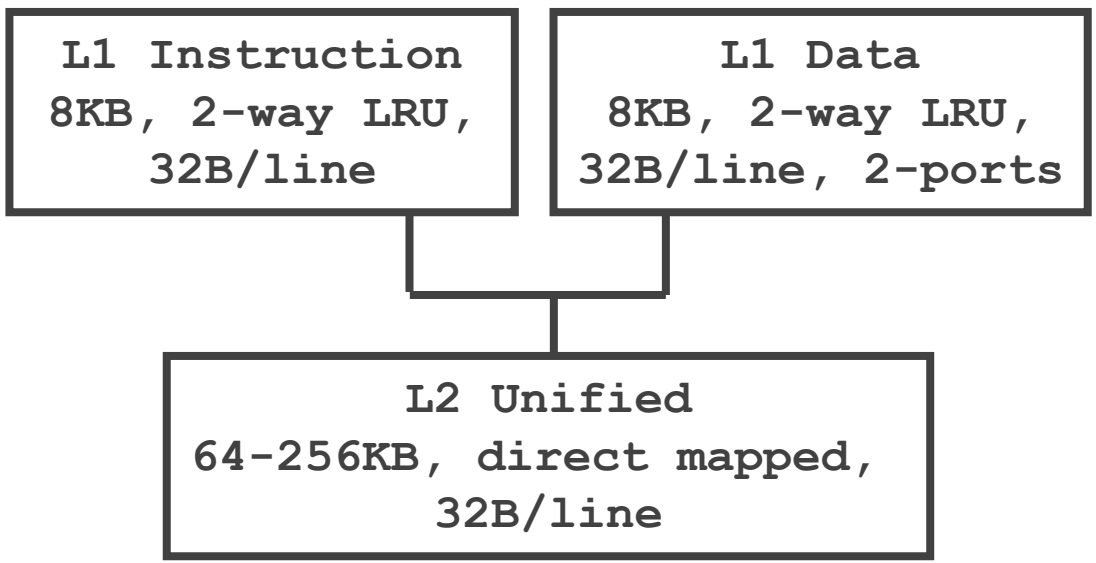
- Το πιο πετυχημένο σχήμα μέχρι στιγμής: Hu, Καξίρας, Martonosi, ISCA 2002
- Correlating prefetcher με πρόβλεψη χρόνου
  - Διατηρεί ιστορία από ακολουθείες tag που εμφανίζονται στη cache:



  - Αν δούμε A,B κάνουμε prefetch το C
  - Προβλέπουμε επίσης πόσο θα μείνει το B στην cache ώστε να κάνουμε prefetch το C μετά τη "ζωή" του B
  - Με 8KB ιστορία ελαχιστοποιούμε το miss rate μιας 32KB L1D σχεδόν όσο είναι θεωρητικά δυνατό

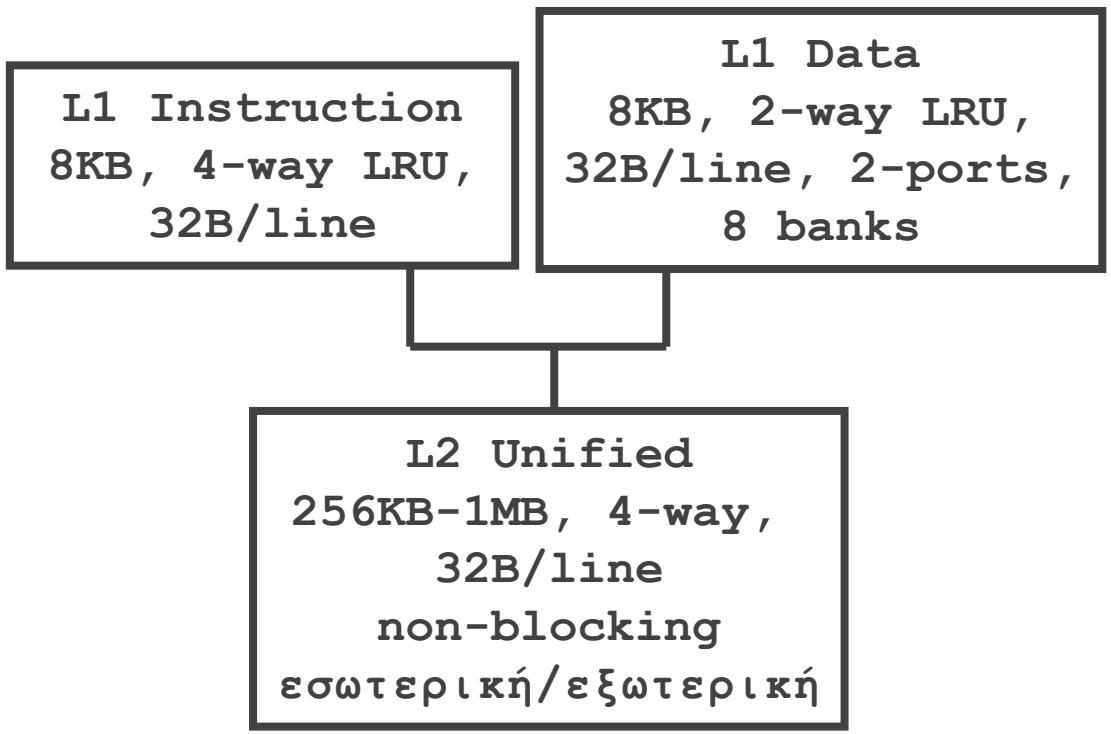
Ιεραρχία Μνήμης στους x86

- 80386 εξωτερική, unified, direct-mapped, 32KB
- 80486 εξωτερική, unified, direct-mapped, 64KB
- Pentium



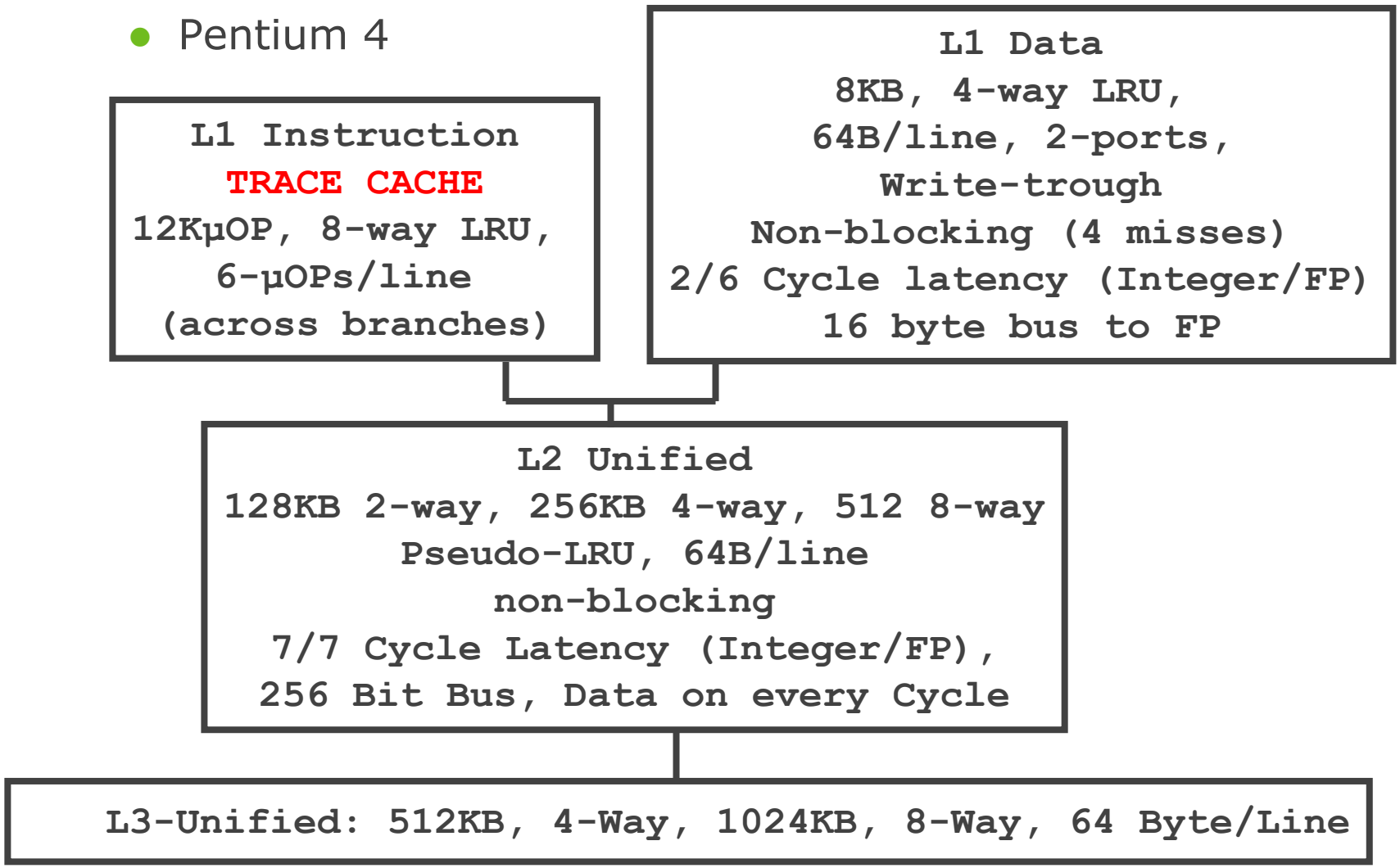
Ιεραρχία Μνήμης στους x86

- Pentium Pro



Ιεραρχία Μνήμης στους x86

- Pentium 4



Κατανάλωση Ισχύος

- Το "καυτό" πρόβλημα της αρχιτεκτονικής σήμερα
  - Intel CTO Pat Gelsinger: "πυκνότητα ισχύος" σε επεξεργαστές αυξάνεται εκθετικά
- Δυναμική ισχύς (λειτουργία τρανζίστορ, ~90%), στατική ισχύς (ρεύματα διαρροής, ~10%)
  - Για να μειώσουμε τη δυναμική ισχύ μειώνουμε την τάση τροφοδοσίας (V_{dd}) => εκθετική αύξηση της στατικής κατανάλωσης ισχύος
- Andy Grove (13/12/2002) : Το τέλος του νόμου του Moore!
  - Λιθογραφία? Κβαντικά φαινόμενα? Κόστος εργοστασίων?
  - Ρεύματα διαρροής

Κατανάλωση Στατικής Ισχύος και μνήμες cache

- CACHE DECAY: Καξίρας, Hu, Martonosi, ISCA 2001
- Διακοπή τάσης τροφοδοσίας σε γραμμές της cache που δεν πρόκειται να χρησιμοποιηθούν ξανά
  - Temporal Locality: Ότι προσπελάστηκε πρόσφατα είναι πιθανό να προσπελαστεί ξανά
  - Ότι δεν έχει προσπελαστεί εδώ και πολύ ώρα είναι απίθανο να προσπελαστεί ξανά
- Μέθοδο να μετράμε χρόνο μεταξύ προσπελάσεων
- Μπορούμε να "σβήσουμε" το 80% της cache (L1) χωρίς σχεδόν καμία επίπτωση στην απόδοση (L2 καλύτερα)
  - δεν σημαίνει ότι μια μικρή cache δουλεύει το ίδιο καλά …
- 5x μείωση κατανάλωσης στατικής ισχύος (αμελητέα αύξηση στη δυναμική ισχύ)

Credits

- Από μνήμης: ιστορικά
- Από μνήμης: Hennessy & Patterson
- Αναζήτηση: www.google.com, www.citeseer.com
- www.sandpile.org (σωρός από άμμο -- πυρίτιο) x86 info
- Sohi & Moshovos “Microarchitectural Innovations: Boosting Microprocessor Performance Beyond Semiconductor Technology Scaling”
- Yale Patt “Requirments, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution”