

# Προηγμένη Σχεδίαση και Σύνθεση Ψηφιακών Συστημάτων

Λεντάρης Γεώργιος

Εργαστήρια και Ειδικά Θέματα

ICE-8205, UNIWA 2025

# Vivado (βλ. αρχείο-οδηγίες στο eclass)

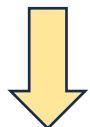
αρχεία VHDL  
(κύκλωμα μου)



synthesis  
(παίρνω netlist)

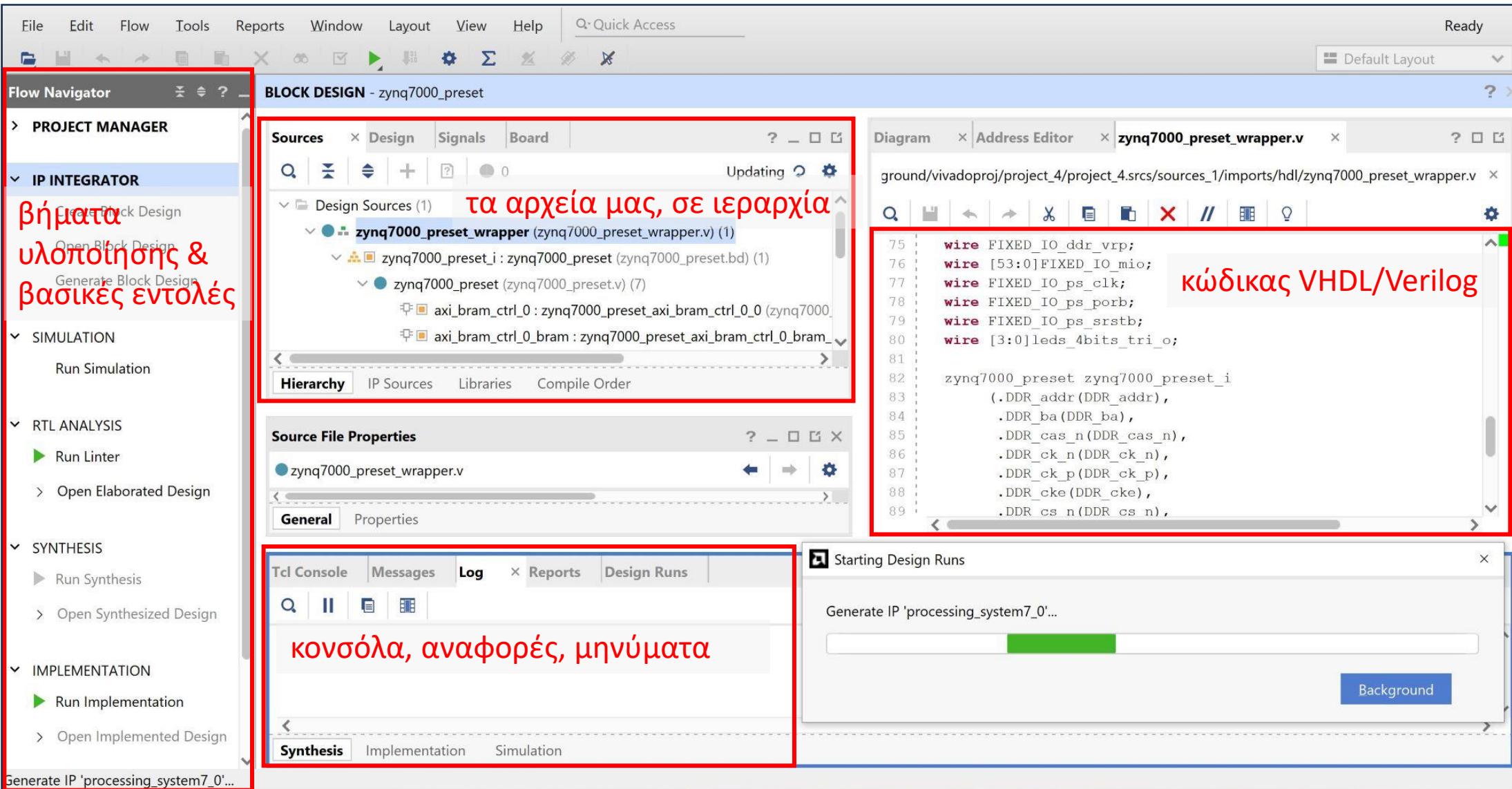


P&R (netlist)



προγρ. FPGA  
(bitstream)

+ simulation !



The screenshot shows the Vivado interface with several windows open:

- Flow Navigator:** Shows the project structure with "IP INTEGRATOR" selected. A red box highlights the "IP INTEGRATOR" section, and handwritten text in red says "βήματα υλοποίησης & βασικές εντολές" (steps of implementation & basic commands).
- BLOCK DESIGN - zynq7000\_preset:** Shows the design sources hierarchy. A red box highlights the "Sources" tab, and handwritten text in red says "τα αρχεία μας, σε ιεραρχία" (our files, in hierarchy). The hierarchy tree includes "zynq7000\_preset\_wrapper", "zynq7000\_preset\_i", and "zynq7000\_preset".
- Diagram:** Shows the VHDL code for "zynq7000\_preset\_wrapper.v". A red box highlights the code area, and handwritten text in red says "κώδικας VHDL/Verilog".
- Tcl Console:** Shows the synthesis process. A red box highlights the "Log" tab, and handwritten text in red says "κονσόλα, αναφορές, μηνύματα" (console, messages, notifications). The log shows "Generate IP 'processing\_system7\_0'...".
- Starting Design Runs:** Shows the progress of the IP generation. A red box highlights the progress bar, which is green.

# Ασκήσεις

# Άσκηση 1: Εξοικείωση, ALU

- Βρείτε/φτιάξτε την ALU του MIPS σε VHDL (ICE-4005 ή αλλού)

[α] υλοποιήστε στο Vivado (στο Zynq, synthesis+implementation)

  - δείτε το resource utilization μετά από κάθε βήμα, συγκρίνετε

[β] αυξομειώστε το μέγεθος του datapath, επαναλάβετε το [α]

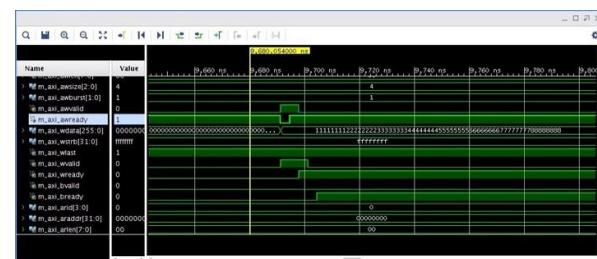
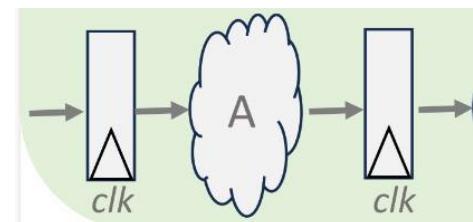
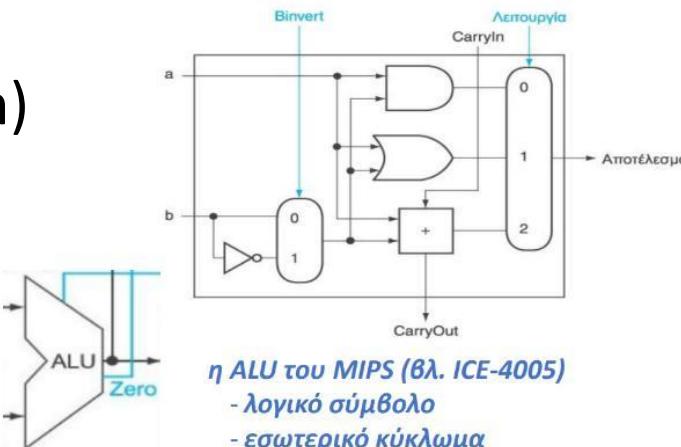
  - π.χ., από 32-bit σε 64- ή 16-bit άθροιση/AND/OR
  - προτιμήστε να δουλέψετε με "generics" στη VHDL

[γ] τοποθετείστε καταχωρητές εμπρός και πίσω από την ALU

  - ώστε να δημιουργήσετε 1 στάδιο σωλήνωσης (το EX του 5-stage MIPS)
  - προτείνεται ιεραρχική σχεδίαση, δηλαδή νέο "top-level" VHDL αρχείο στο οποίο θα βάλετε σαν components την ALU και τους καταχωρητές

[δ] προσομοιώστε την νέα σας μονάδα με το Vivado Simulator

  - πρώτα "behavioural", μετά "post-implementation functional"
  - Ανοίγει μέσα από το Vivado. Σημειώνεται ότι θα χρειαστεί να φτιάξετε ένα testbench για το νέο σας VHDL top-level αρχείο (βλ. ICE-4005)



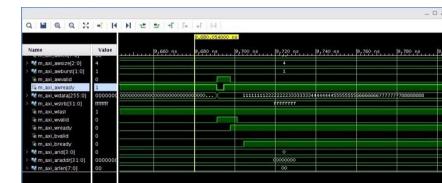
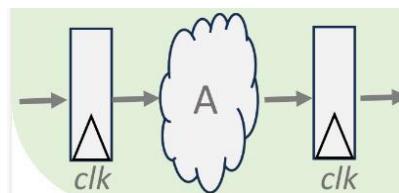
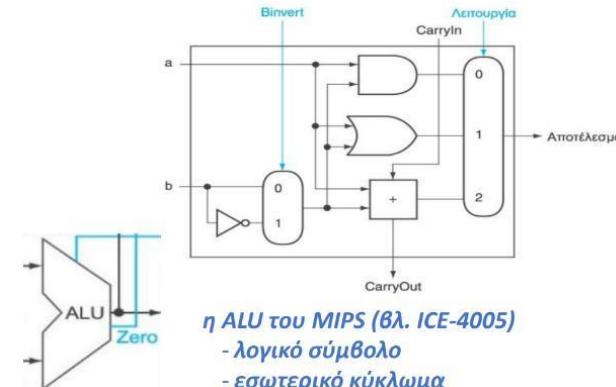
# Άσκηση 1: Εξοικείωση, ALU

- Βρείτε/φτιάξτε την ALU του MIPS σε VHDL (ICE-4005 ή αλλού)

- [ε] προχωρείστε σε Timing Analysis (μετά το implementation)
- δείτε πόσο γρήγορη είναι η ALU σας (τη μέγιστη συχνότητα ρολογιού)
  - Θα χρειαστεί να βάλτε Xilinx Timing Constraints και να "πιέσετε" το εργαλείο να κάνει καλύτερη υλοποίηση (βλ. και settings synthesizer)
  - επαναλάβετε και συγκρίνετε το αποτέλεσμα για 32- και 64-bit ALU

- [ζ] βελτιώστε τη συχνότητα λειτουργίας προσθέτοντας DFF
- βάλτε ένα ακόμα στάδιο καταχωρητών κάπου μέσα στην ALU (είναι σχεδιαστική επιλογή δική σας), δηλαδή, φτιάξτε 2-stage ALU pipeline
  - πόσο είναι το καινούριο throughput και το καινούριο latency της ALU?
  - θέλουν αλλαγές τα σήματα ελέγχου για να λειτουργήσει σωστά η ALU?

- [η] δείτε το Power Analysis του FPGA design που έχετε φτιάξει
- παρατηρήστε τα static και dynamic power, αλλάζει με τη συχνότητα  $f_{clk}$  ?
  - **ΕΠΙΠΡΟΣΘΕΤΑ:** βάλτε και πολλαπλασιαστή στην ALU, δείτε πόρους/συχνότητα (αποφύγετε DSP blocks, θεωρείστε I/O=32-bits) →



Θεωρείστε προσημασμένους αριθμούς στο (-1,1), κρατείστε μόνο τα MSB στην έξοδο

# Άσκηση 2: FSM και stream processing

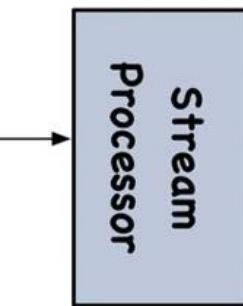
- υλοποιήστε σε FPGA μια μικρή μονάδα επεξεργασίας ροής δεδομένων, μέσω χρήσης Finite State Machine. Η λειτουργικότητα να είναι ως εξής:

[1] παρακολουθεί μια ροή δεδομένων εισόδου και τα μεταβάλει ανάλογα με το περιεχόμενο τους (πρωθεί στην έξοδο, ενδεχομένως αλλαγμένα)

- σημείωση1: τα δεδομένα είναι 8-bit, εισέρχονται σειριακά σε συνεχόμενους κύκλους, η έξοδος τους επιτρέπεται να καθυστερήσει κατά 2 κύκλους (latency=2)
- σημείωση2: η ροή εισόδου-εξόδου δεν πρέπει να σταματάει ποτέ (η μονάδα μας θα προωθεί πάντα κάτι στην έξοδο)

[2] αν σε οποιαδήποτε χρονική στιγμή εντοπίσει το δεδομένο "0" μέσα στη ροή εισόδου, τότε τα επόμενα 10 δεδομένα θα υποστούν προσαύξηση της τιμής τους κατά "+1" (πχ, "...8,3,1,0,1,1,2,5,3,3..." → "...8,3,1,0,2,2,3,6,4,4...")

- σημειώση3: αυτό συμβαίνει ΚΑΘΕ φορά που εντοπίζεται "0". Αν το "0" βρίσκεται μέσα σε ήδη ανιχνευμένη δεκάδα, τότε η τρέχουσα δεκάδα επεκτείνεται αναλόγως



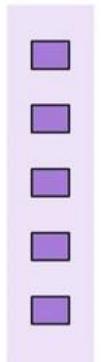
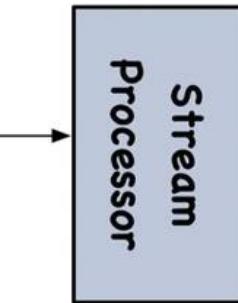
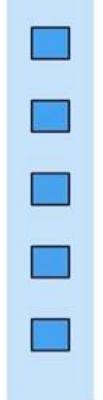
# Άσκηση 2: FSM και stream processing

- υλοποιήστε σε FPGA μια μικρή μονάδα επεξεργασίας ροής δεδομένων, μέσω χρήσης Finite State Machine. Η λειτουργικότητα να είναι ως εξής:

[3] αν σε οποιαδήποτε χρονική στιγμή εντοπίσει το **δεδομένο "255"** στη ροή εισόδου, τότε μόνο για το 3<sup>o</sup> δεδομένο που ακολουθεί θα πρέπει να διπλασιαστεί η τιμή του (πχ, "8,3,255,1,1,2,2,3..." --> "8,3,255,1,1,4,2,3...")

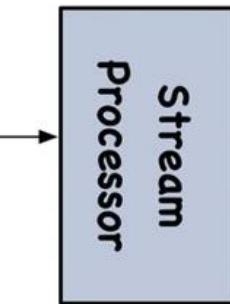
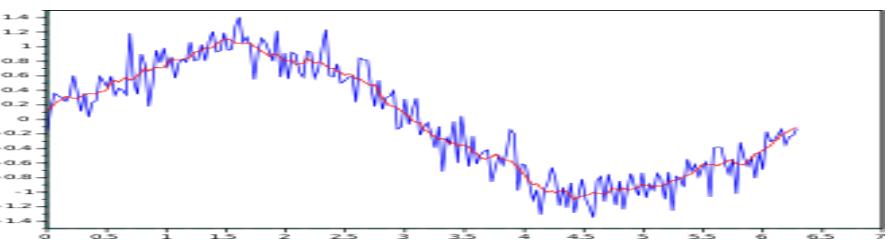
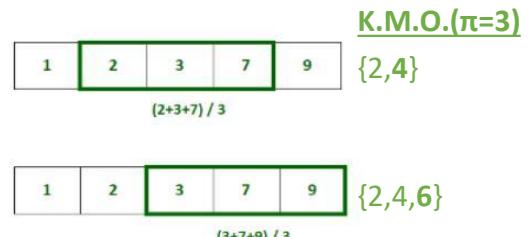
- **σημείωση4:** το 1<sup>o</sup>+2<sup>o</sup> δεδομένο δεν επηρεάζουν τη ροή, ακόμα κι αν είναι 0 ή 255
- **σημείωση5:** οι προσαυξήσεις/διπλασιασμοί οδηγούν σε "clipping", δηλαδή οποιοδήποτε αποτέλεσμα μεγαλύτερο του 255 πρωθείται στην έξοδο ως 255
- **σημείωση6:** η διαδικασία προσαύξησης "+1" έχει προτεραιότητα, δηλαδή, δεν λαμβάνεται υπόψη τυχόν "255" που βρίσκεται εντός της δεκάδας που ακολουθεί το "0"

- *Γράψτε HDL, αλλά πρώτα σχεδιάστε την αρχιτεκτονική σε σχηματικό στο χέρι*
- *Ελέγξτε με testbench, σε behavioral και post-implementation functional simulation*
- *Προτείνεται FSM με 3 ξεχωριστές καταστάσεις για την περίπτωση διπλασιασμού, αλλά μόνο 1 ξεχωριστή κατάσταση και μετρητή για την περίπτωση προσαύξησης.*



# Άσκηση 2: FSM και stream processing

- υλοποιήστε σε FPGA μια μικρή μονάδα επεξεργασίας ροής δεδομένων.
- **ΕΠΙΠΡΟΣΘΕΤΑ:** υλοποιήστε κινητό μέσο όρο (σε δεύτερη ξεχωριστή έξοδο)
  - υποθέστε πλάτος παραθύρου 8 τιμών (Μ.Ο. τελευταίων 8 εισόδων, sliding window)
  - υλοποιήστε χωρίς/εκτός του κυρίως FSM (άλλο component, απλά έχουν κοινή είσοδο)
  - **σημείωση A:** Θα χρειαστείτε κάποιου είδους μνήμης 8 τιμών, προτείνεται shift-register
  - **σημείωση B:** για λόγους βελτιστοποίησης, βασιστείτε σε συσσωρευτή (accumulator) που αφαιρεί την 8η παρελθοντική τιμή (π.χ., αντί για παράλληλο αθροιστή 8 τιμών)
  - **σημείωση Γ:** υποθέστε θετικούς ακέραιους σε είσοδο-εξόδο (8-bit unsigned), προσοχή στην αφαίρεση και στις αναπαραστάσεις (θέλουμε βέλτιστη χρήση διαθέσιμων bit)
  - **σημείωση Δ:** η διαίρεση με  $2^k$  σε υλικό γίνεται με "διαγραφή/εξαφάνιση" των  $k$  LSB\*
  - γράψτε HDL, αλλά πρώτα να σχεδιάσετε την αρχιτεκτονική σε σχηματικό (στο χέρι)



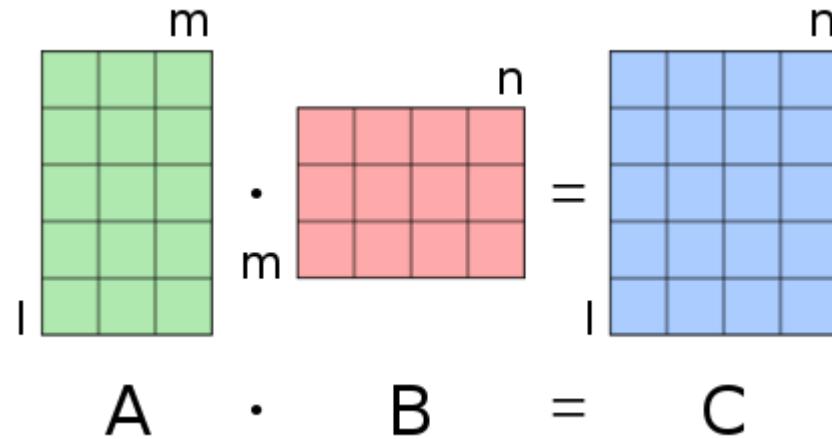
\*LSB=Least Significant Bits

# Εργασίες

(διαλέγουμε 1)

# Πολλαπλασιασμός Πινάκων

- μεταβλητό μέγεθος, έως πίνακες 64x64
  - Θα κατασκευαστούν μονάδες/μνήμες που υποστηρίζουν το μέγιστο μέγεθος 64x64, ελέγχουμε μέρος που χρησιμοποιούμε
- αριθμ. σταθερής υποδιαστολής 16-bit
  - προσημασμένοι '2Σ' στο διάστημα (-1,1)
  - μετά τις πράξεις επανερχόμαστε στα 16b
- υποθέτουμε είσοδο κι έξοδο σε RAMBs
  - γεμίζουμε από/σε FPGA pins, πχ, σειριακά
- εφαρμόζουμε pipelining
- μελετάμε αρχιτεκτονικές
  - υλοποιούμε & συγκρίνουμε



$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

$$\mathbf{C} = \begin{pmatrix} a_{11}b_{11} + \cdots + a_{1n}b_{n1} & a_{11}b_{12} + \cdots + a_{1n}b_{n2} & \cdots & a_{11}b_{1p} + \cdots + a_{1n}b_{np} \\ a_{21}b_{11} + \cdots + a_{2n}b_{n1} & a_{21}b_{12} + \cdots + a_{2n}b_{n2} & \cdots & a_{21}b_{1p} + \cdots + a_{2n}b_{np} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{11} + \cdots + a_{mn}b_{n1} & a_{m1}b_{12} + \cdots + a_{mn}b_{n2} & \cdots & a_{m1}b_{1p} + \cdots + a_{mn}b_{np} \end{pmatrix}$$

# Πολλαπλασιασμός Πινάκων

- **αρχιτεκτονικές**

- Σειριακή

- 1 μονάδα Multiply-Acc. (έως 64 κύκλους για 1  $c_{ij}$ )
- 1 FSM, με χρήση 1 MAC για όλα τα  $n \cdot l$  στοιχεία C

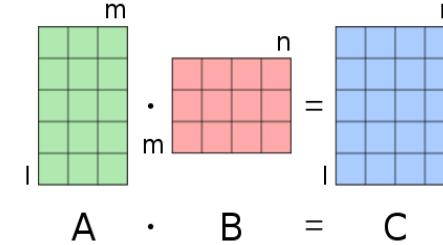
- Παράλληλη ως προς Διάνυσμα·Διάνυσμα

- $m$  πολλαπλασιαστές και δέντρο αθροιστών
- pipelined (9+2 στάδια συνολικά, μαζί με το I/O)
- παράλληλη μνήμη: διαβάζω  $2m$  data σε 1 κύκλο
  - πχ,  $m$  RAMB για στήλες A,  $m$  RAMB για γραμμές B
  - αλλιώς χάνεται το πλεονέκτημα παραλληλίας "Δ·Δ"
- σειριακή χρήση 1 "Δ·Δ" για όλα τα  $n \cdot l$  στοιχεία C

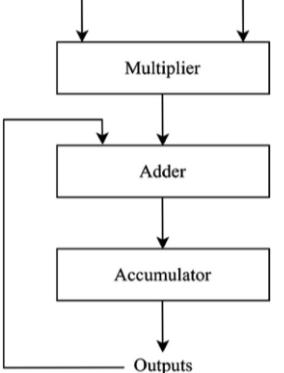
- Παράλληλη ως προς ξεχωριστά στοιχεία  $c_{ij}$

- $n \cdot l$  μονάδες MAC σε διάταξη 2D mesh, με 1 FSM
- ή πολλές μονάδες "Δ·Δ" **(πόσο κλιμακώνεται!?)**

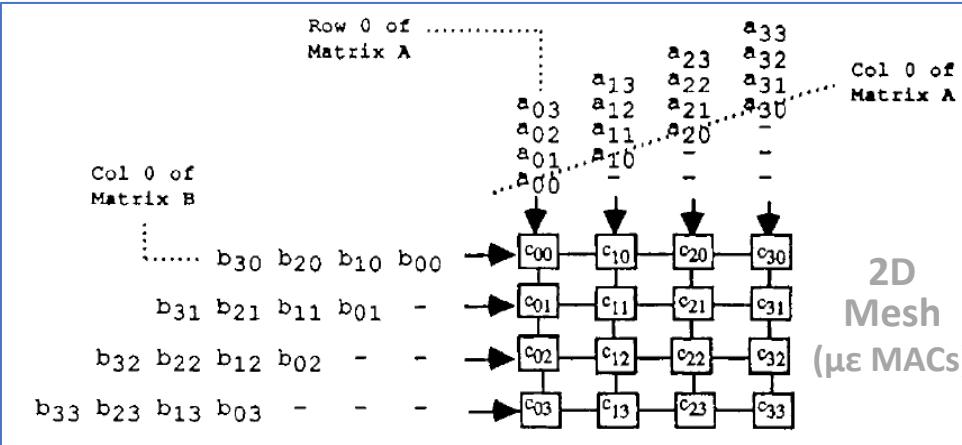
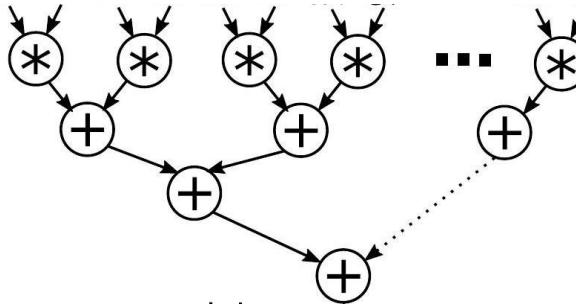
- **μελετάμε** πόρους, ρυθμαπόδοση, καθυστέρηση,  $f_{max}$



σειριακό MAC



παράλληλο MAC (mult-accumulate)

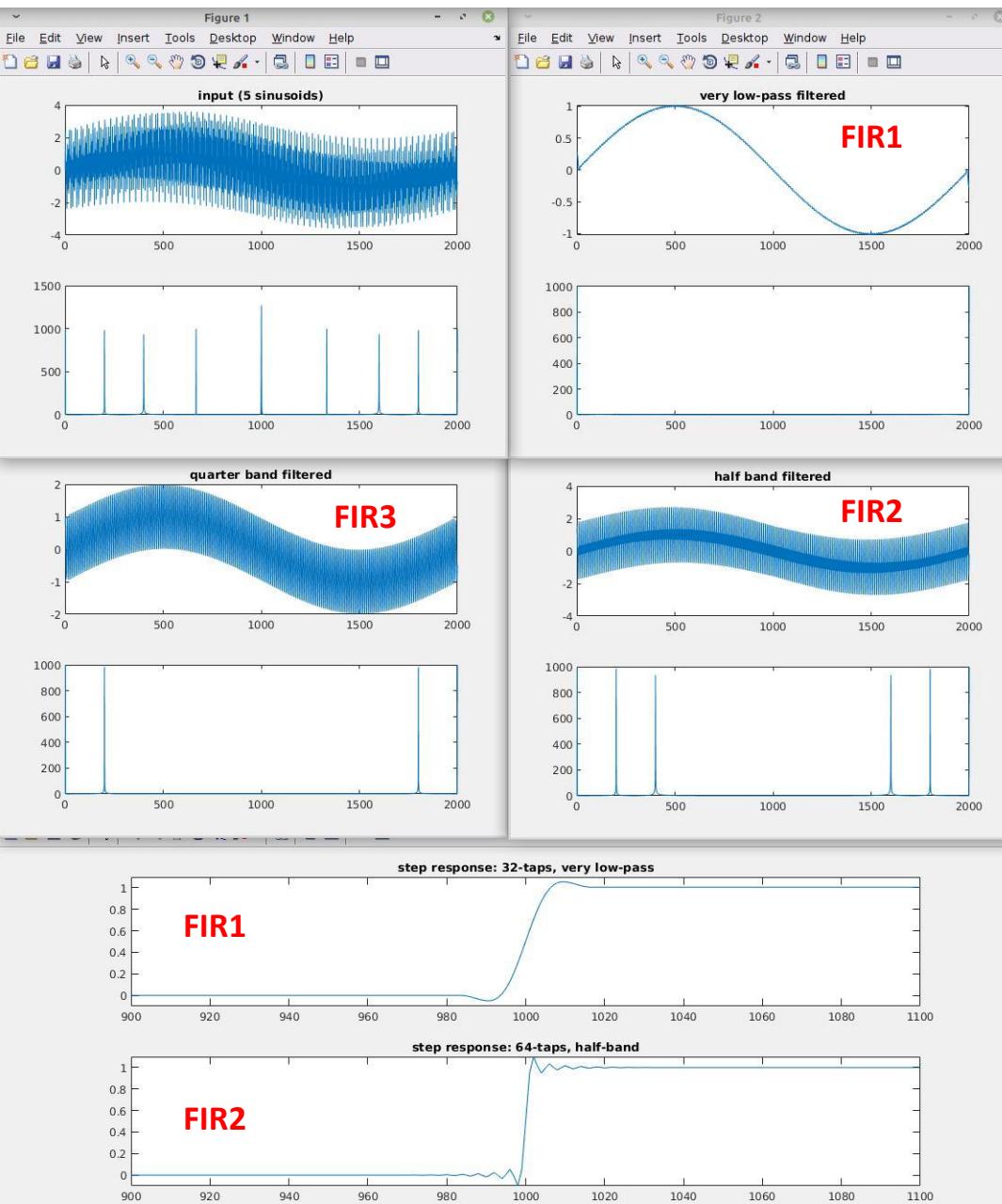
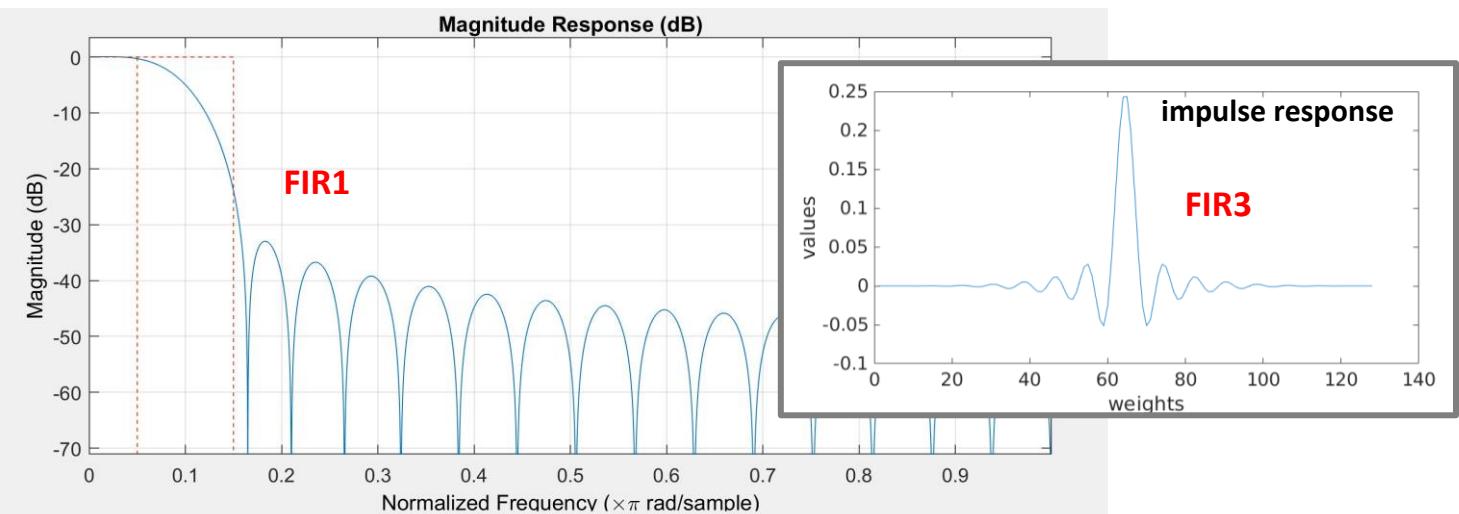


βελτιστοποιούμε αριθμό σταδίων pipeline &  $f_{max}$ , data routing

κάνουμε έλεγχο με testbench και αντιπροσωπευτικά δεδομένα

# Φίλτρο FIR (*finite impulse response*)

- 1D, πραγματικό, χαμηλοπερατό
  - taps = 32 ή 64 ή 128 (cutoff= 90% / 50% / 75%)
  - αριθμ. σταθερής υποδιαστολής (16bit)
    - weights  $\in (-0.1, 0.5)$  , I/O data  $\in (-1, 1)$
- φόρμουλα (είσοδος  $x[n]$ , έξοδος  $y[n]$ ,  $taps=N$ )
 
$$y[n] = b_0x[n] + b_1x[n - 1] + \dots + b_Nx[n - N]$$



# Φίλτρο FIR (*finite impulse response*)

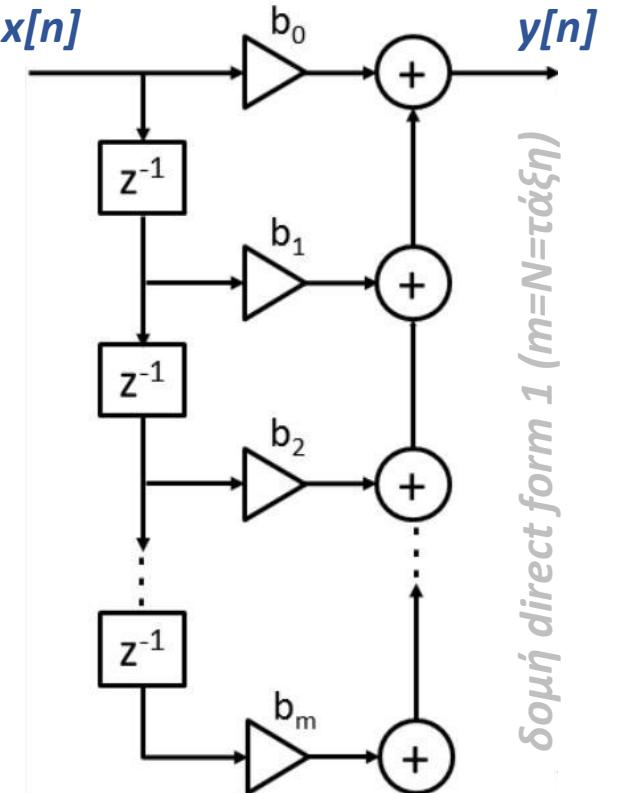
$$y[n] = b_0x[n] + b_1x[n - 1] + \cdots + b_Nx[n - N]$$

- **αρχιτεκτονική**

- πλήρως παράλληλη
- ακολουθεί *direct form 1*
- pipelined (συν I/O registers)
- σε κάθε κύκλο θα εισάγεται νέο δεδομένο (16-bit I/O)  
(εξάγεται μετά από  $N$  κύκλους)
- σήματα enable, valid in/out

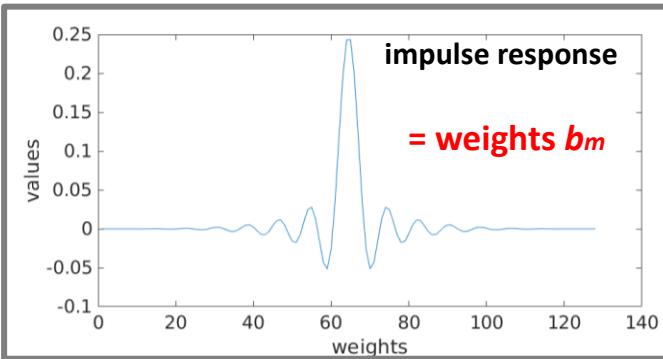
- **μελέτη** (ενδεικτικά)

- σειριακή vs παράλληλη υλοποίηση (βλ. πόρους,  $f_{max}$ )  
(σειριακή\*: 1  $x[n]$  ανά  $N$  κύκλους)
- FIR1 vs FIR2 vs FIR3 (πόροι,...)



**σημ.: αρχιτεκτονική κυκλώματος όχι ταυτόσημη με direct form 1...**

- πως υλοποιώ καθυστερήσεις  $z^{-1}$ ?
- πως υλοποιώ τεράστια άθροιση?
- συστολική γραμμή, δέντρο?



**σημ.: Θα βρείτε έτοιμα βάρη  $b_m$  στο eclass (.zip, στο φάκελο εργαστήρια)**

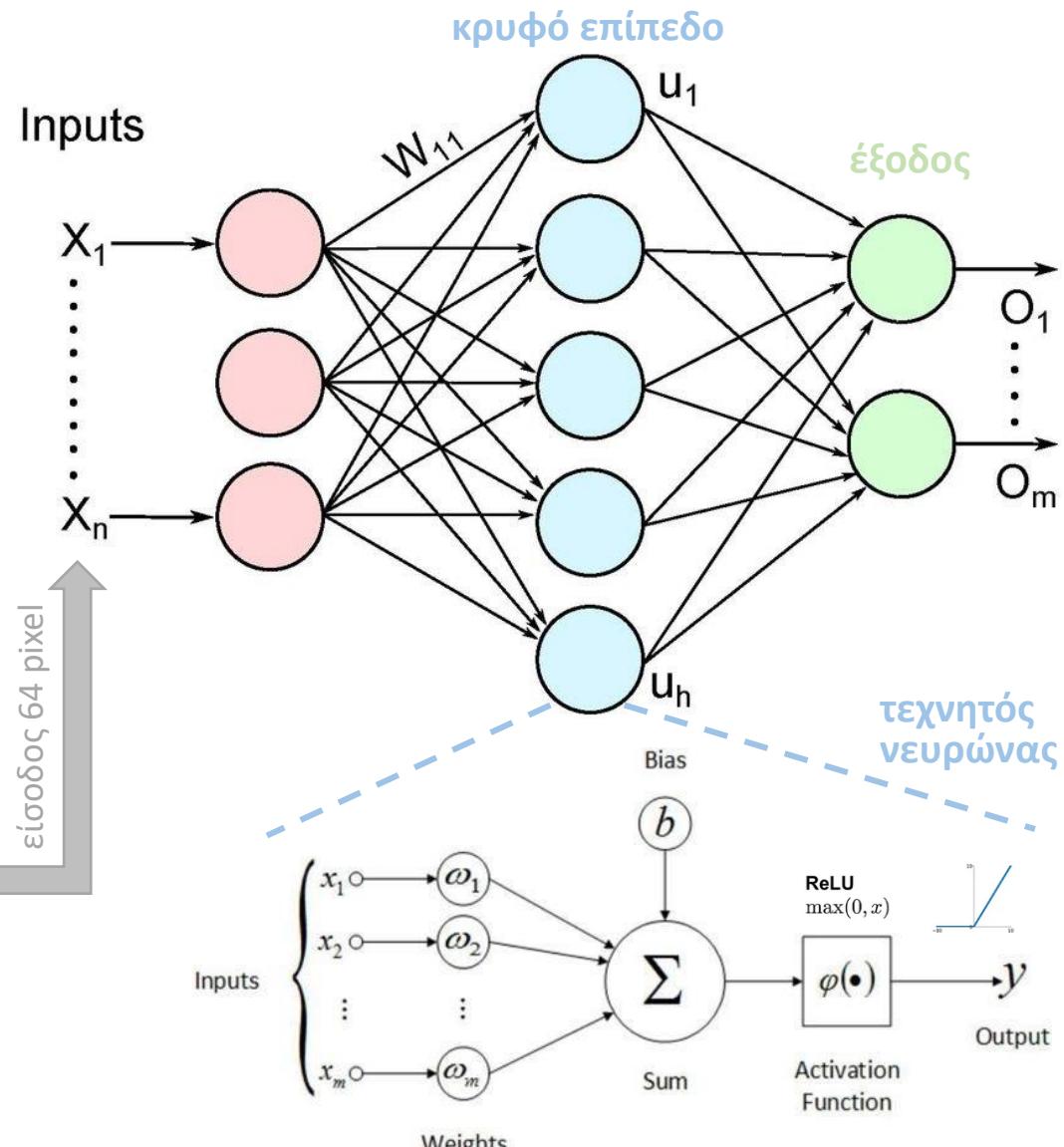
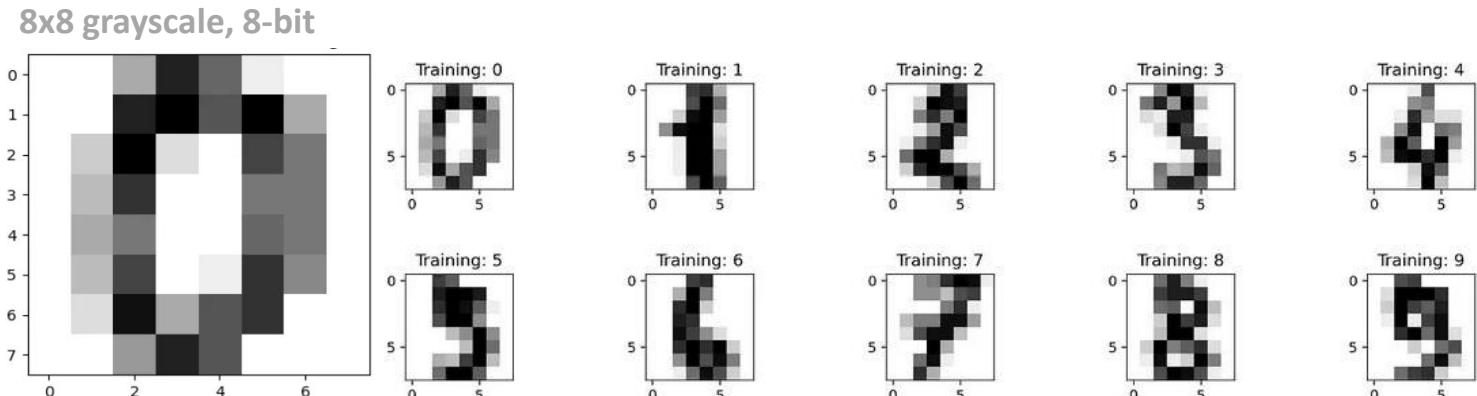
- FIR1 (32taps), FIR2 (64taps), FIR3(128t)
- να τα μετατρέψετε σε fixed-point 2Σ πριν χρησιμοποιήσετε σε HDL: **Q1.15**
  - στο (-1,1), πχ, "0.1011..." (16b)

## βελτιστοποιήσεις (ενδεικτικά)

- συμμετρικά βάρη? γίνεται με τους μισούς πολλαπλασιστές.
- σειριακό? επαναχρησιμοποίηση πόρων (1 MULT-ACC, 1 RAMB)
- με ή χωρίς DSP blocks?
- ...άλλες?

# Multi-Layer Perceptron (AI)

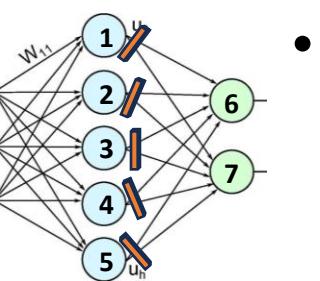
- αναγνώριση χαρακτήρων σε εικόνα
- με MLP δίκτυο ενός κρυφού επιπέδου
  - **64 είσοδοι** (εικόνα σε raster-scan, 2D $\mapsto$ 1D)
  - **32 κρυφοί νευρώνες** (fully-connected, ReLU)
    - άρα έχουμε:  $32 \times 64$  βάρη  $w_{ij}$  και 32 biases  $b_i$
    - κάθε νευρώνας πολλ/ζει κι αθροίζει εισόδους
  - **10 έξοδοι** (10 κλάσεις: χαρακτήρες '0', '1', ..., '9')
    - 10 νευρώνες:  $10 \times 32$  βάρη  $w_{ij}$  και 10 biases  $b_i$
    - σημ.: δεν θα βάλουμε "soft-max" στο τέλος



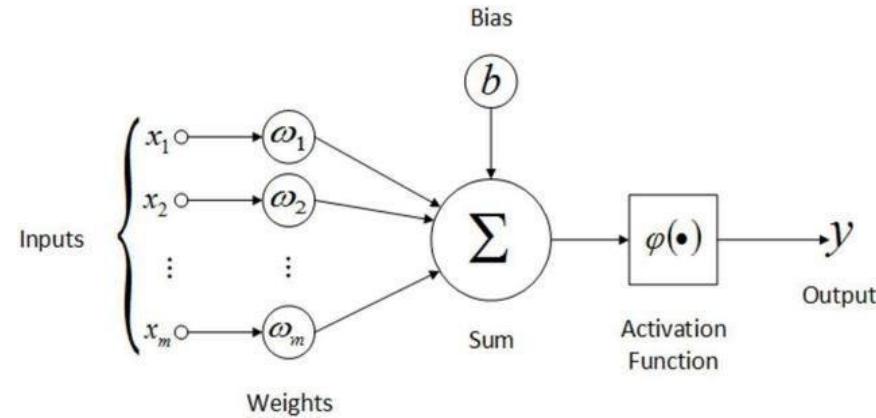
# Multi-Layer Perceptron (AI)

## • αρχιτεκτονική

- αριθμητική: σταθερής υποδιαστολής
  - είσοδος 8bit [0,255], βάρη 9bit Q1.8  $\in (-1,1)$ , εσωτερικές πράξεις 14bit Q6.8  $\in [-32,32]$
- νευρώνας: πλήρως παράλληλος
  - 64 πολλαπλασιαστές (για κρυφό νευρώνα)
  - δέντρο αθροιστών, μετά συγκριτής (ReLU)
  - pipeline 8+2 σταδίων (συνολικά, με I/O)
- δίκτυο MLP: σειριακά τους κόμβους, με FSM
  - 1 μηχανή τρέχει όλους τους νευρώνες, έναν-έναν
  - αποθηκεύουμε τα βάρη σε ROM 65 παράλληλων εξόδων, σε κάθε κύκλο τροφοδοτεί νέο νευρώνα
    - μέγεθος ROM: depth=(32+10)x65, width=9bit
- είσοδος-έξοδος: υποθέτουμε παράλληλη
  - όλα τα  $X$  μαζί, με σήμα 'start' (στον ίδιο κύκλο)
  - όλα τα  $O$  μαζί, με σήμα 'end'  η έξοδος  $O_n$  με την μεγαλύτερη τιμή μας δείχνει την επικρατέστερη από τις 10 κλάσεις (0,1,...9)



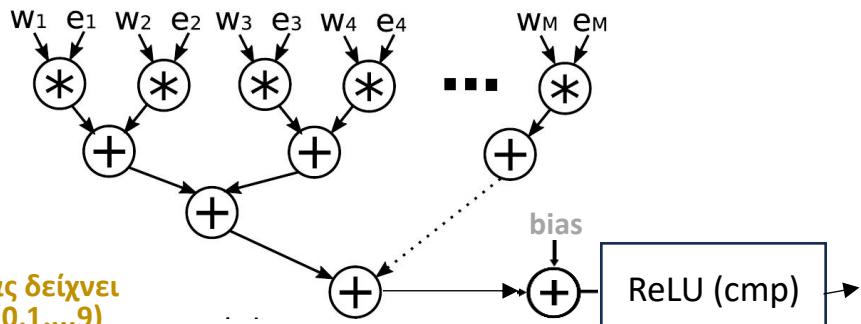
Σειριακά 1-5,  
αποθήκευση  
ενδιάμεσων  
αποτελεσμάτων  
σε **registers** για  
παράλληλο διάβασμα  
κατά την εκτέλεση 6-7



**σημ.:** Θα βρείτε έτοιμα βάρη  $w_{ij}$  στο eclass (.zip, στο φάκελο εργαστήρια)

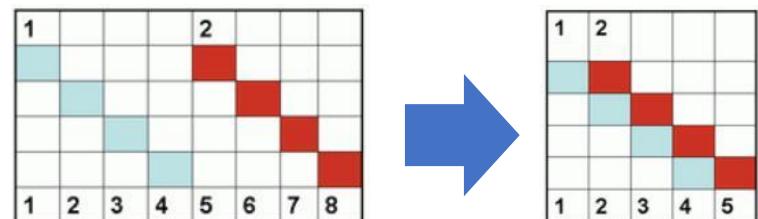
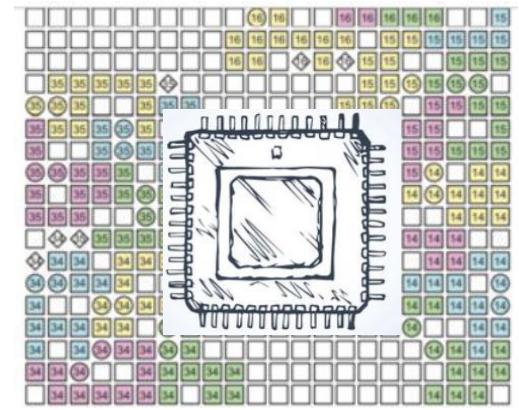
- για 32+10 νευρώνες (κρυφοί, έξοδοι)
- να τα μετατρέψετε σε fixed-point 2S πριν χρησιμοποιήσετε σε HDL: **Q1.8**
  - στο (-1,1), πχ, "0.10110..." (9b)

είσοδος μηχανής-νευρώνα = 64 pixel + 64 weights  
(όλα παράλληλα, από registers και ROM), συν 1 bias



# Σημειώσεις (για όλες τις εργασίες)

- αν δεν χωράνε όλες οι είσοδοι-έξοδοι στα #pins του FPGA που έχουμε διαλέξει (πχ, 484), εφαρμόζουμε πολυπλεξία στο χρόνο
  - πχ, MLP: στον 1<sup>o</sup> κύκλο του 'start' εμφανίζονται τα μισά X (0-31) και στον επόμενο τα υπόλοιπα
  - I/O είναι ενδεικτικό εδώ (όχι κύριο αντικείμενο)
- γενικά, σκοπός είναι να βελτιστοποιήσουμε throughput, με όσο λιγότερους πόρους HW
  - άρα προσέχουμε το χρονοπρογραμματισμό των βημάτων, ώστε να μεγιστοποιήσουμε τη χρήση του υλικού (να μην περιμένει άεργο το pipeline)
- παραδίδουμε κώδικα, τεστ, και καλό report
  - αποφασίζετε βάθος μελέτης/συγκρίσεων/optim.



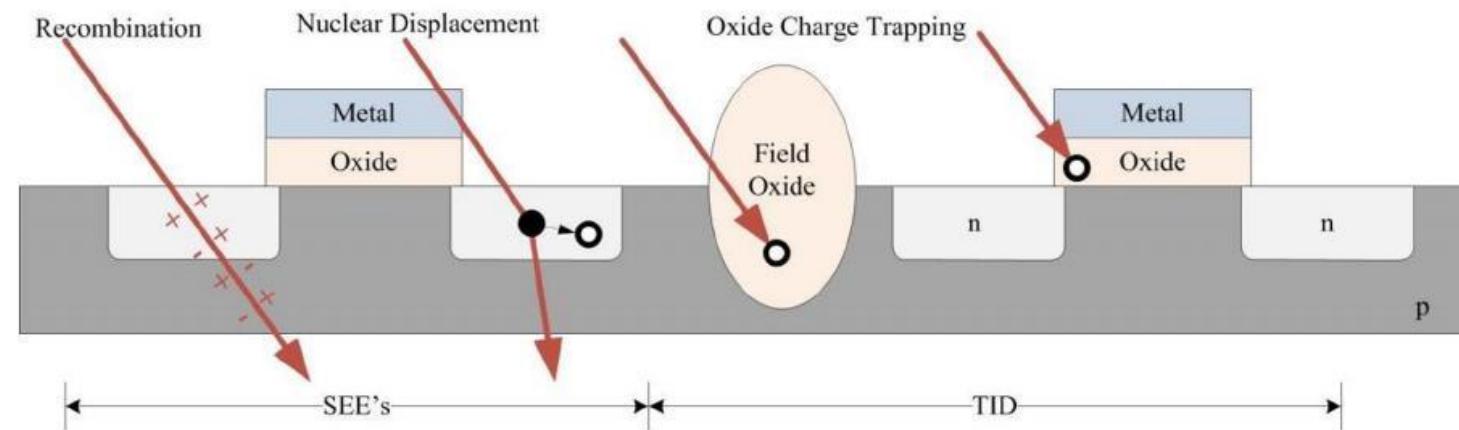
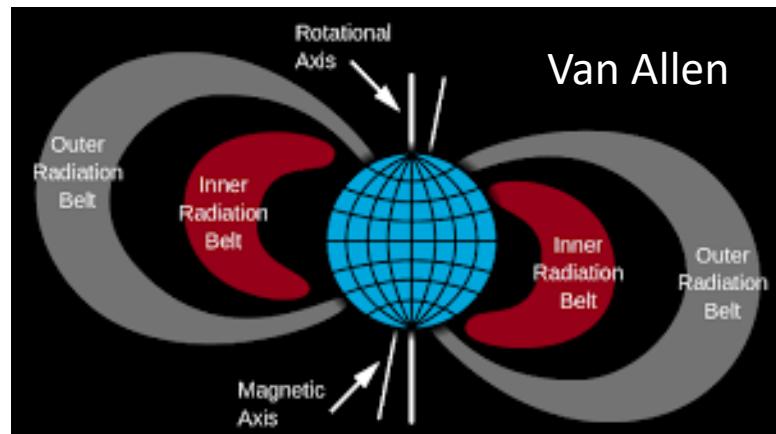
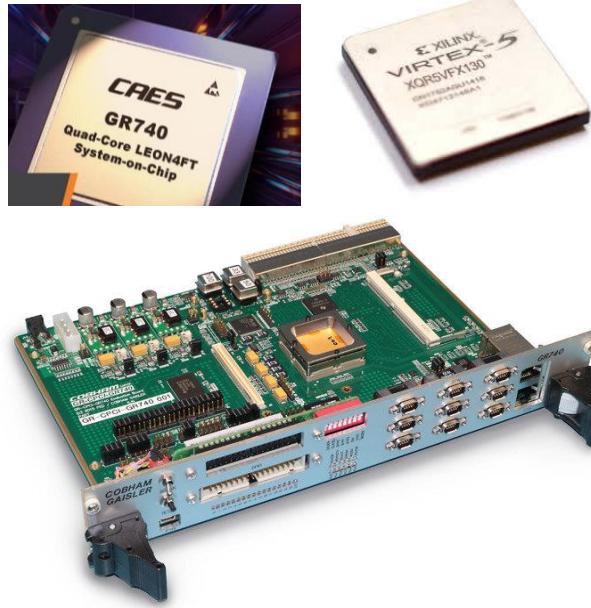
# Παραδείγματα Εφαρμογών

FPGA για Επιτάχυνση Ρομποτικής Όρασης σε Διαστημικές Αποστολές

- Lentaris G., et al. "High-performance vision-based navigation on SoC FPGA for spacecraft proximity operations." *IEEE Trans. on Circuits & Systems for Video Technology* 30.4 (2019): 1188-1202.
- Lentaris G., et al. "Single-and multi-FPGA acceleration of dense stereo vision for planetary rovers." *ACM Transactions on Embedded Computing Systems (TECS)* 18.2 (2019): 1-27.
- Lentaris G., et al. "HW/SW codesign and FPGA acceleration of visual odometry algorithms for rover navigation on Mars." *IEEE Trans. on Circuits & Syst. for Video Tech.* 26.8 (2015): 1563-1577.
- Lentaris G., Stamoulias I., Diamantopoulos D., Maragos K., Siozios K., Soudris D., Rodrigalvarez M.A., Lourakis M., Zabulis X., Kostavelis I. and Nalpantidis L., "SPARTAN/SEXTANT/COMPASS: advancing space rover vision via reconfigurable platforms." *International Symposium on Applied Reconfigurable Computing*. Cham: Springer International Publishing, 2015
- Kostavelis I., Nalpantidis L., Boukas E., M. A. Rodrigalvarez, Stamoulias I., Lentaris G., Diamantopoulos D., Siozios K., Soudris D., Gasteratos A.. "Spartan: Developing a vision system for future autonomous space exploration robots." *Journal of Field Robotics* 31, no. 1 (2014): 107-140.

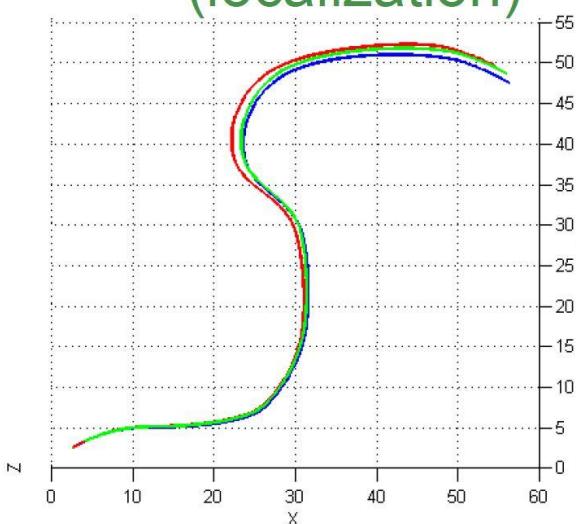
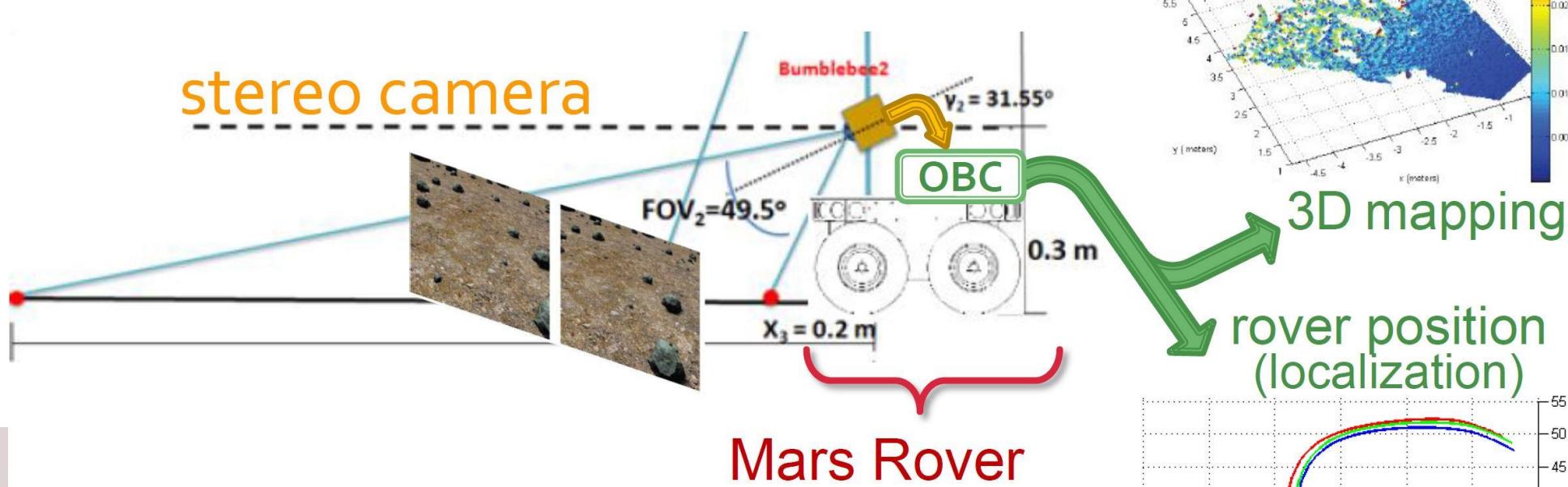
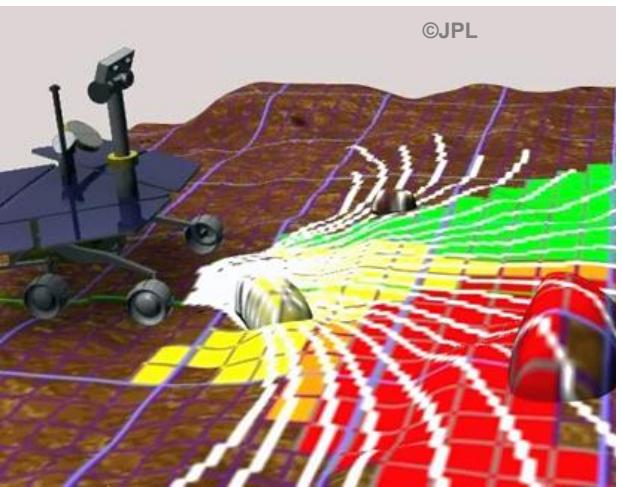
# Γιατί FPGA στο διάστημα? (κι όχι μόνο ASIC ή CPU)

- "μικρή" αγορά, οικονομία κλίμακος ευνοεί και FPGA
  - επίσης: έχει **εξατομικευμένα κυκλώματα**, αρκετή έρευνα
- Ιονίζουσα ακτινοβολία ⇒ σφάλματα στο υλικό/VLSI
  - τεχνικές ενδυνάμωσης VLSI ⇒ κόστος++ ⇒ καθυστέρηση παραγωγής τσιπ (μια δεκαετία πίσω...) ⇒ **αργές RH CPU**
  - FPGA ευνοεί παραλληλισμό ⇒ **επιτάχυνση** (co-processor)
    - επίσης: επιτρέπει **κάποιες τεχνικές ενδυνάμωσης** στο πεδίο



# Εφαρμογή: Visual odometry

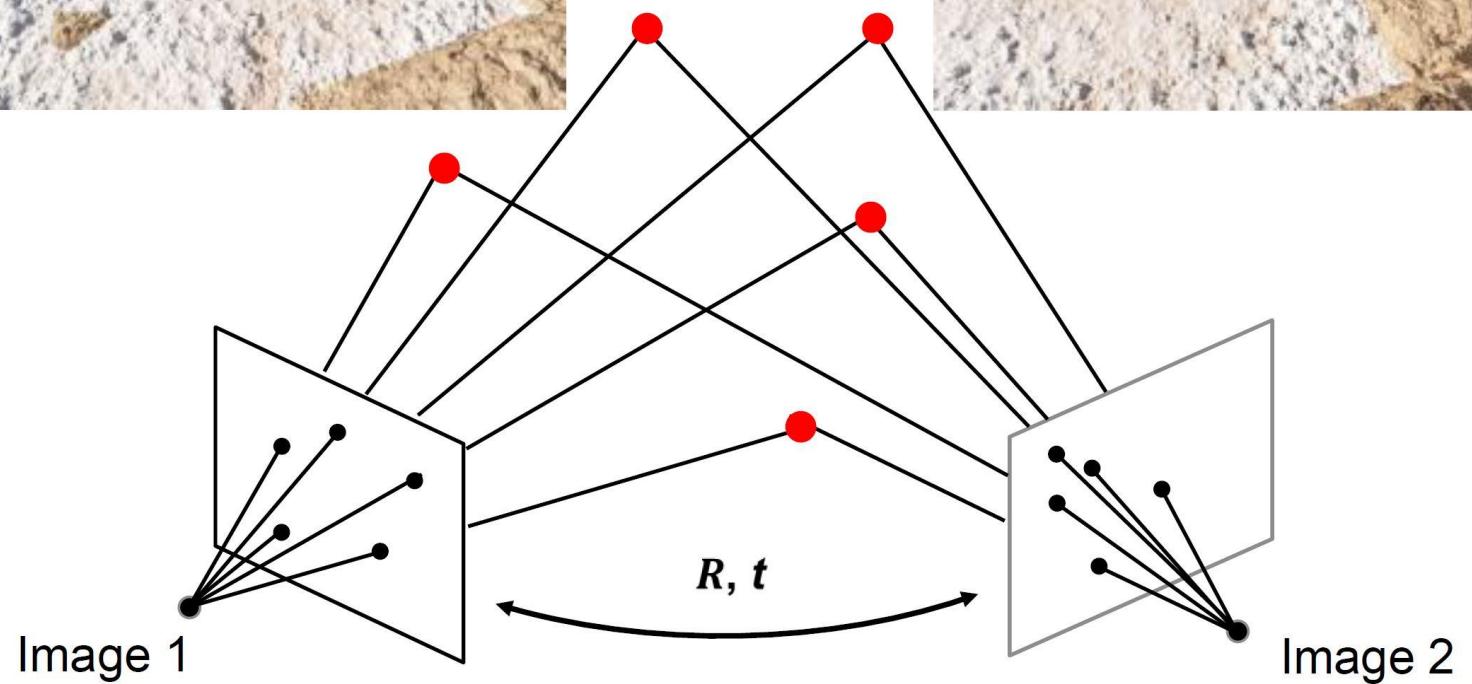
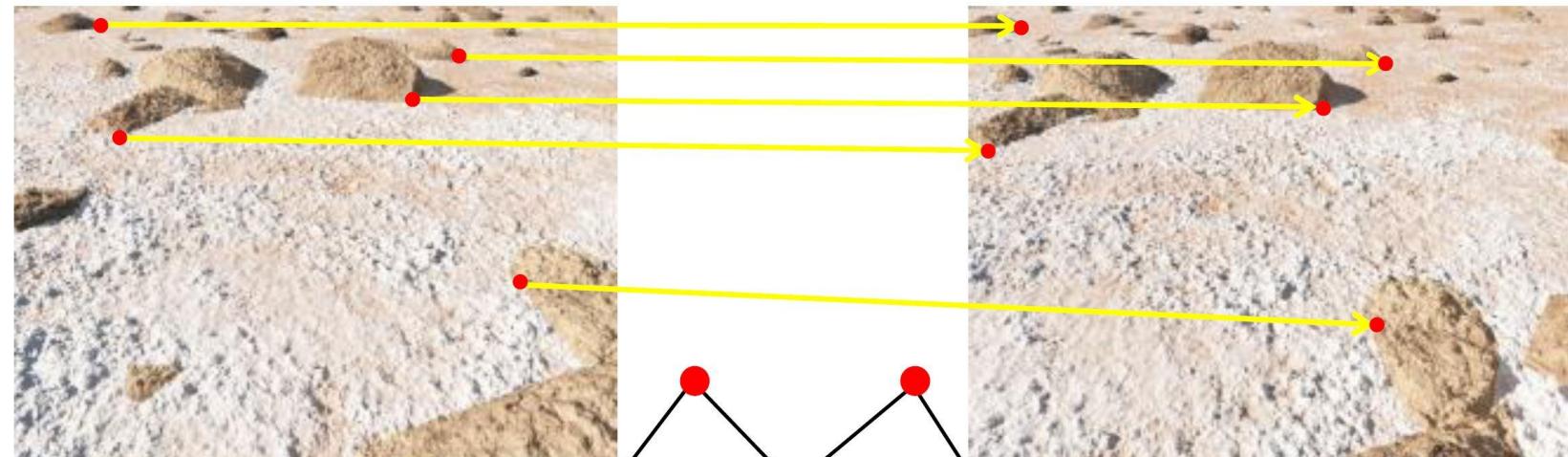
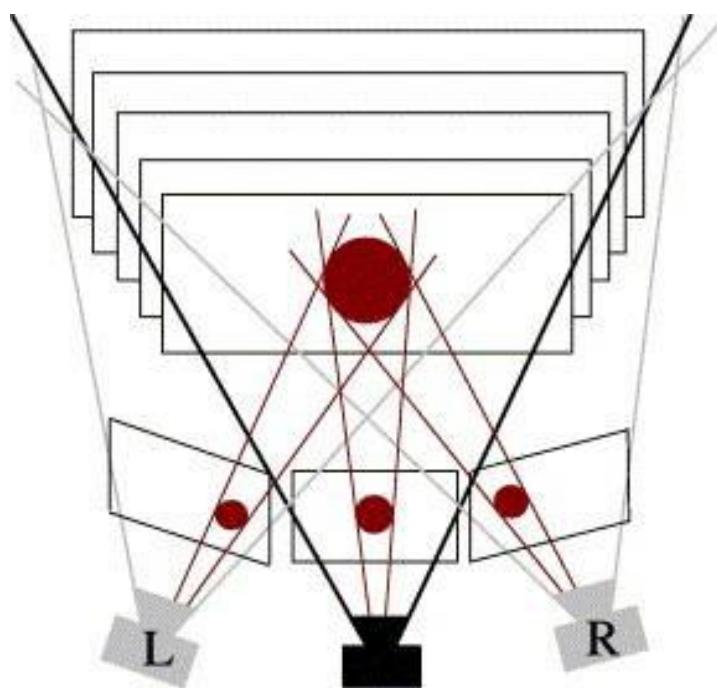
- Mars Rover (κ.α.)
    - προσδιορισμός θέσης στο χώρο
    - ανακατασκευή του 3D χώρου
- VSLAM



# 3D reconstruction

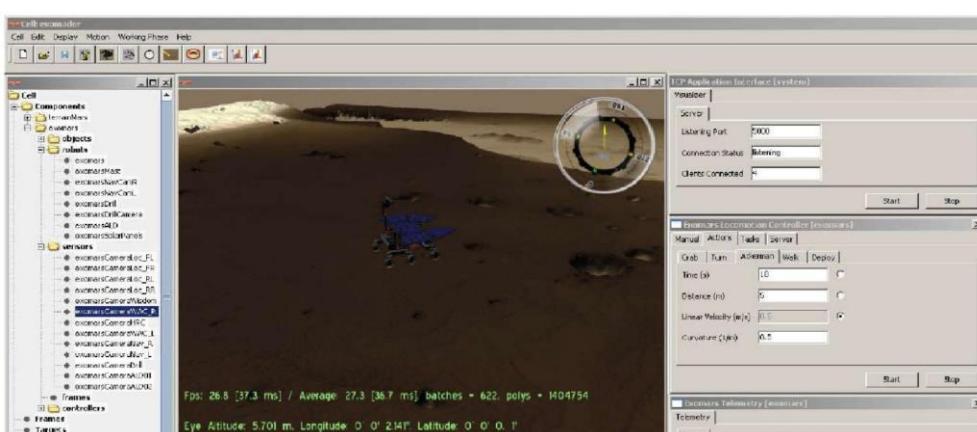
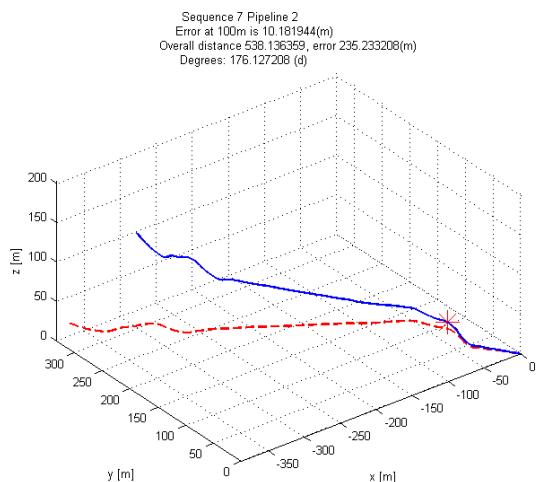
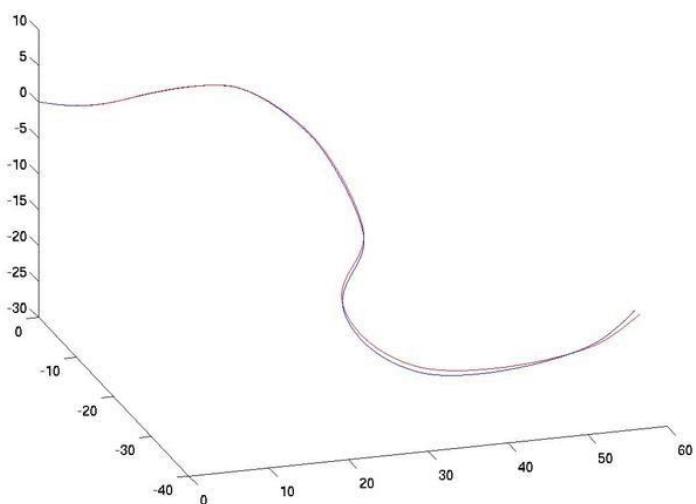


- Disparity-based
- Spacesweep
- K.α.



# Datasets

- συνθετικά
  - ακριβές 'groundtruth'
  - ελεγχόμενη δυσκολία
- πραγματικά
  - πιο αντιπροσωπευτικά (σκιές, φωτισμός,...)
  - απαιτητική συλλογή

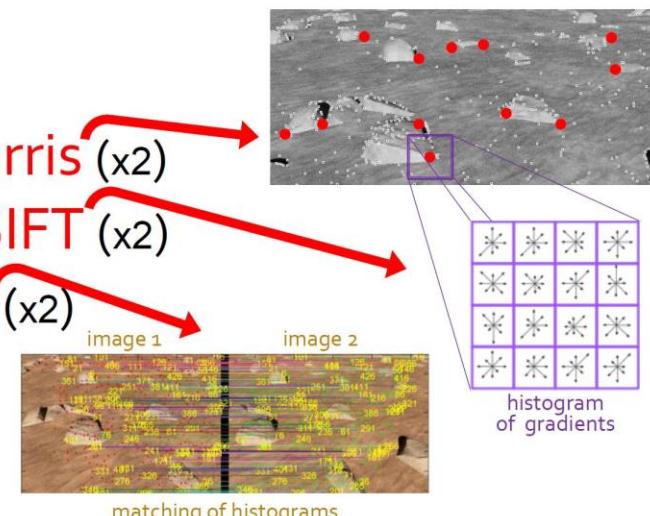


# Αλγόριθμοι

- αλυσίδες αλγορίθμων για διαδοχικά βήματα/προβλήματα (algorithm pipelines)

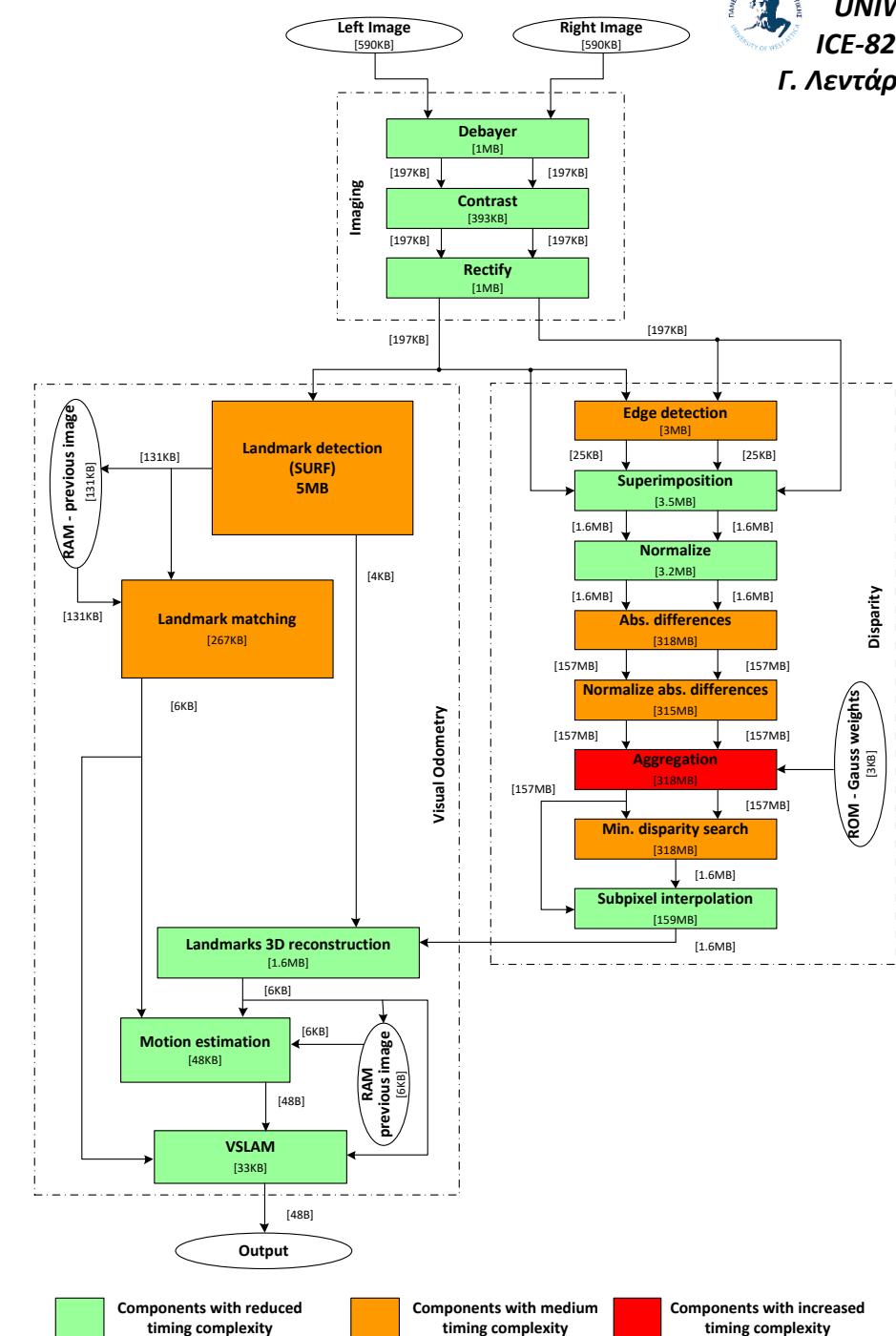
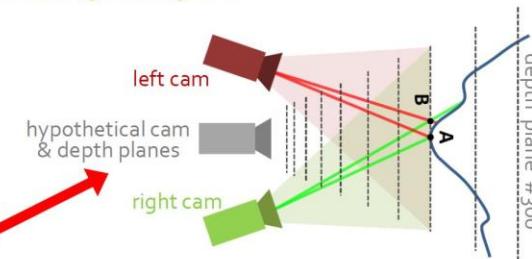
## LOCALIZATION:

1. Feature Detection: **Harris** (x2)
2. Feature Description: **SIFT** (x2)
3. Matching:  **$x^2$ -distance** (x2)
4. Filter outlier matches
5. Motion Estimation



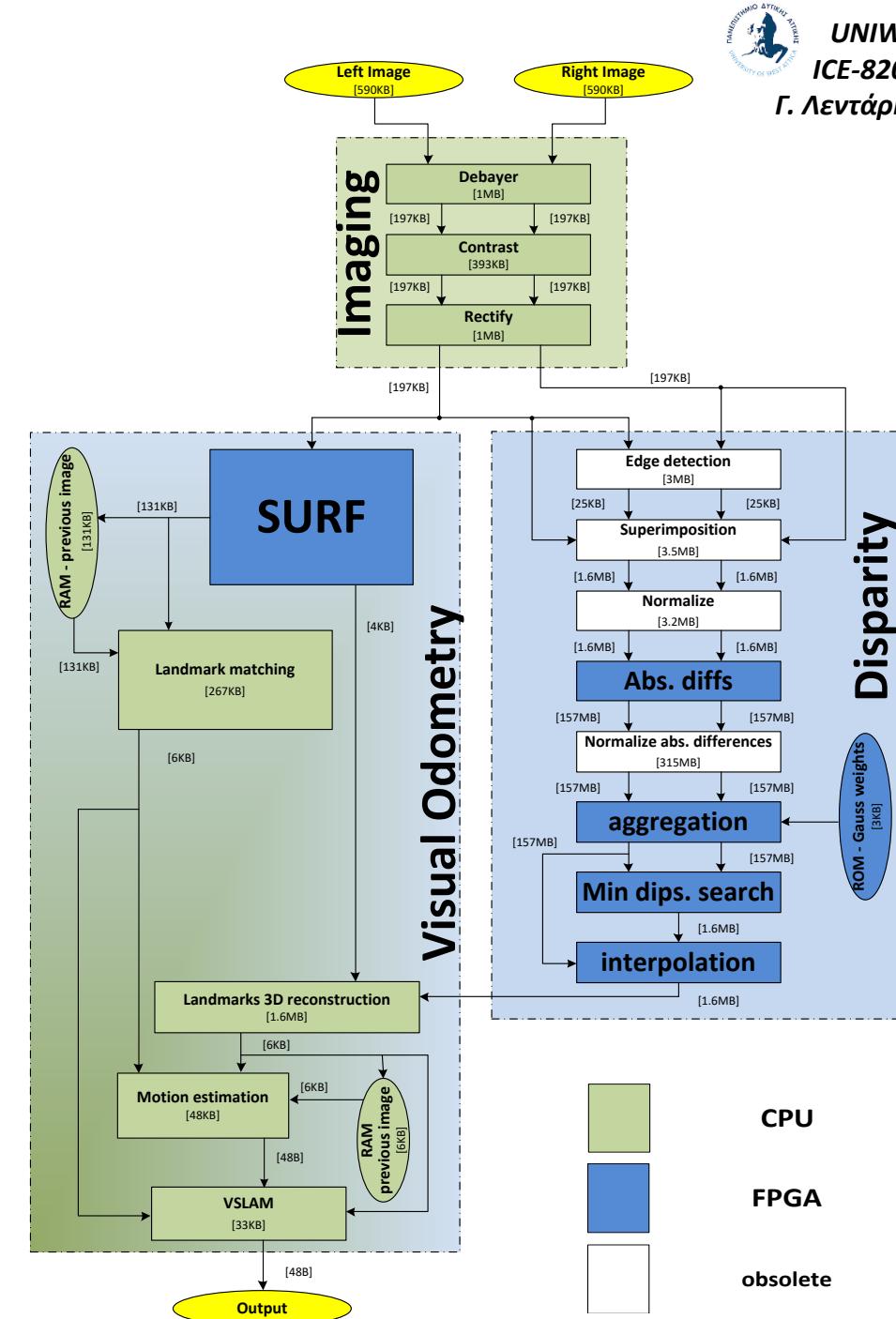
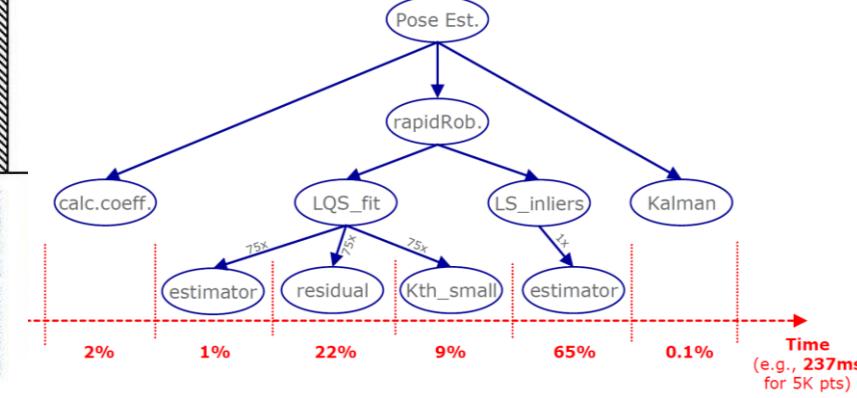
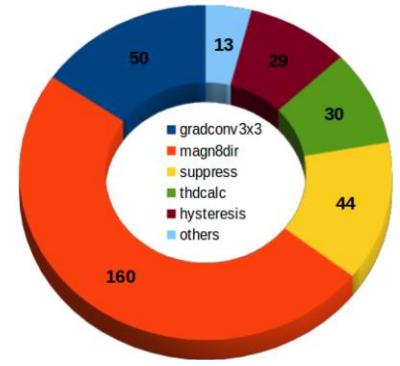
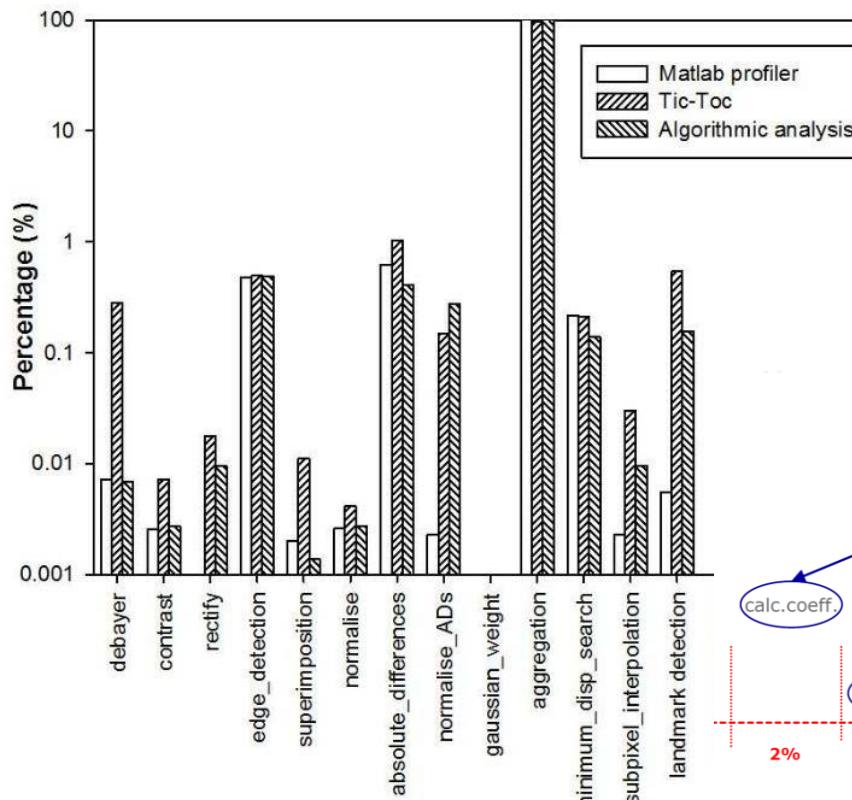
## MAPPING:

brute-force examine 300 depth planes:  
**SpaceSweep**



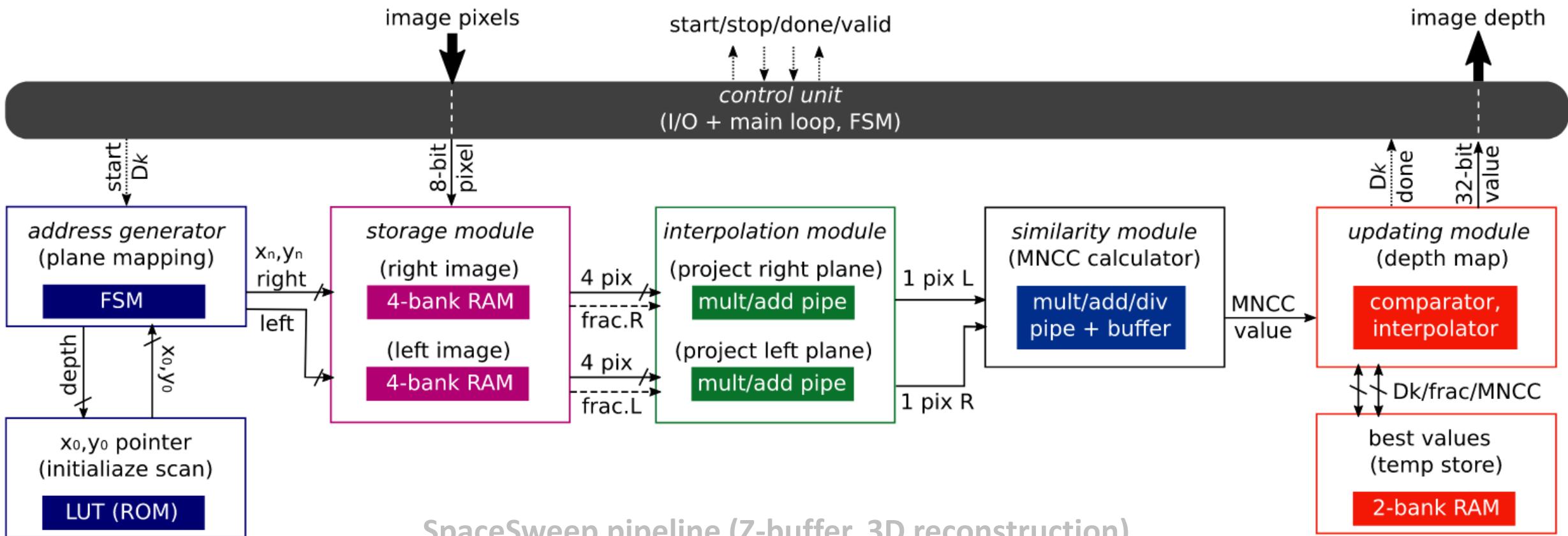
# HW/SW partitioning

- κάποια σε CPU, κάποια σε FPGA. Ποια?
  - μεθοδολογία! profiling, μελέτη αλγορίθμων,...



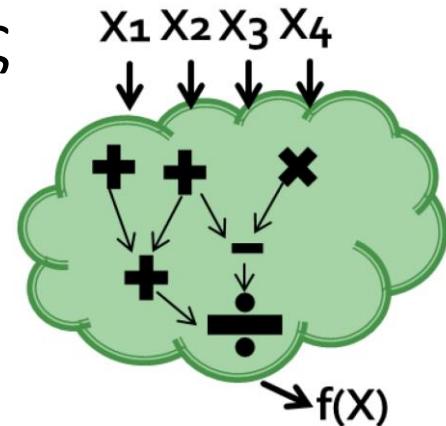
# Κυκλώματα

- deep-pipelining, on-the-fly processing ( $\Rightarrow$  memory min.), distributed control



# Κυκλώματα

- παραλληλισμός αριθμητικών πράξεων
- παράλληλη τροφοδοσία



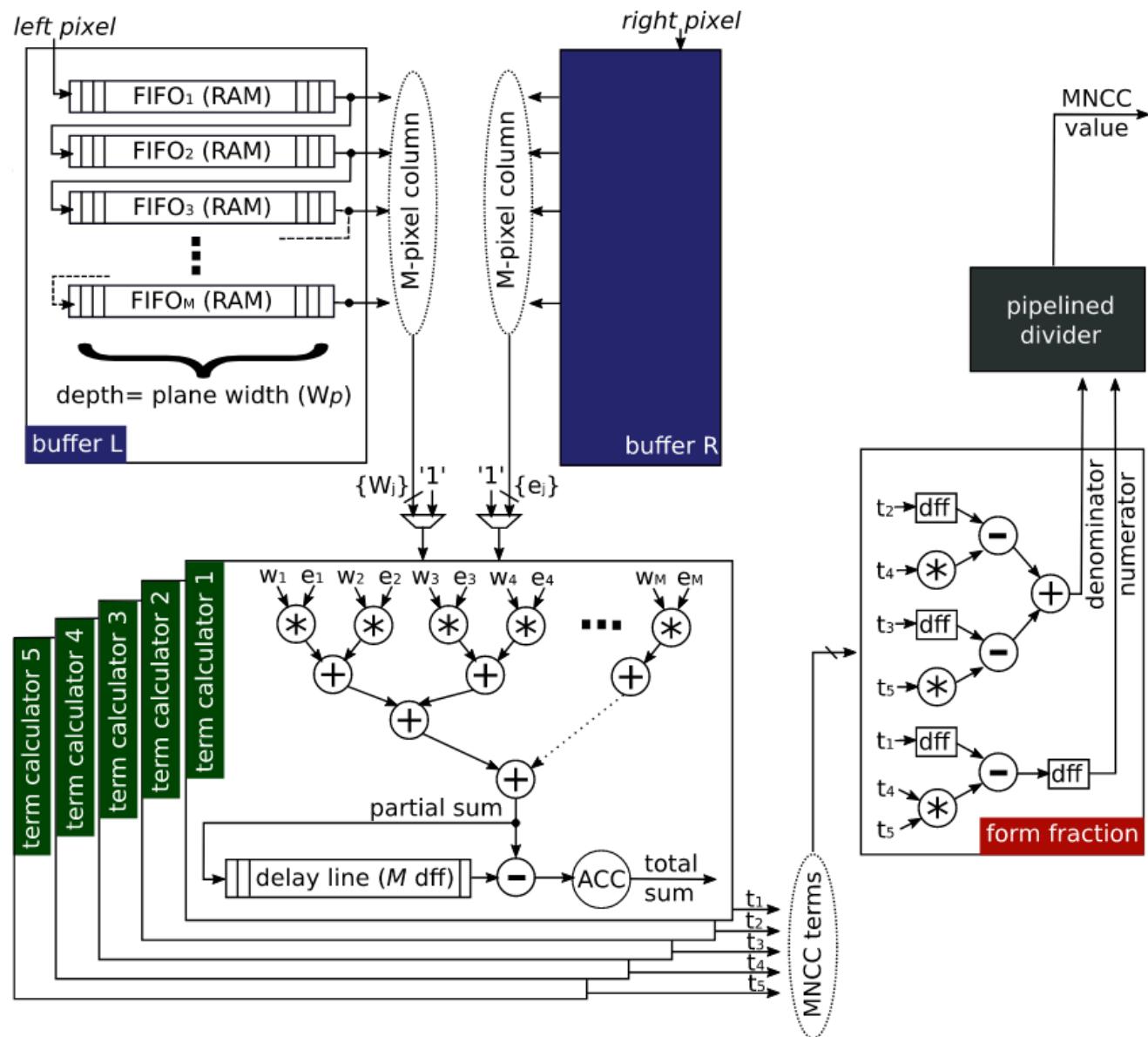
ΠΑΡΑΔΕΙΓΜΑ:  
κανονικοποιημένη ετεροσυσχέτιση εικόνων  $P_1$  vs  $P_2$  (με covariance)

$$\frac{1}{n} \sum_{x,y} \frac{1}{\sigma_f \sigma_t} (f(x,y) - \mu_f) (t(x,y) - \mu_t)$$

$$NCC(P_1, P_2) = \text{cov}(P_1, P_2) / (\sqrt{\sigma^2(P_1) \sigma^2(P_2)})$$

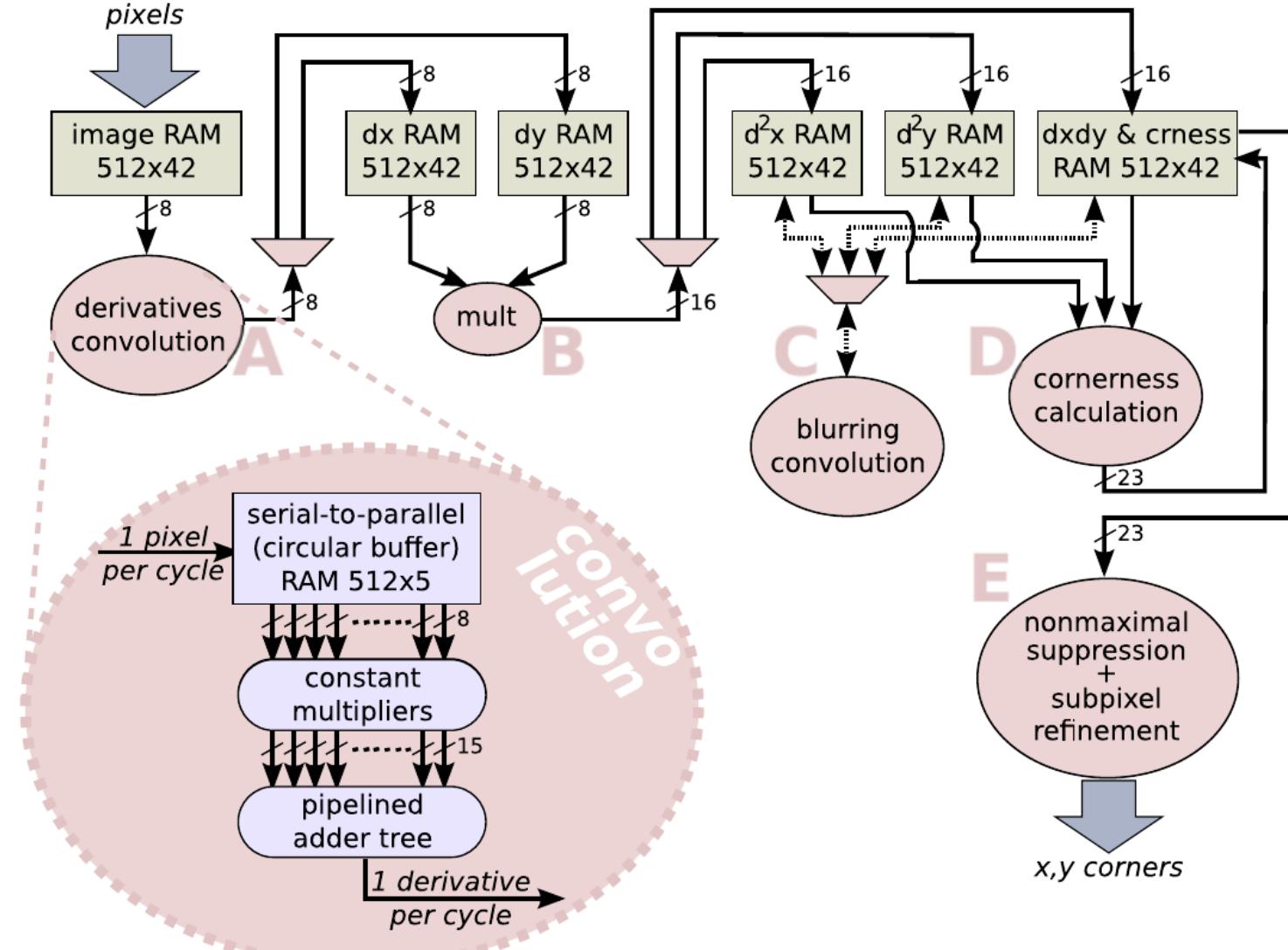
$$MNCC(P_1, P_2) = 2 \text{cov}(P_1, P_2) / (\sigma^2(P_1) + \sigma^2(P_2))$$

$$MNCC(P_1, P_2) = \frac{2(n \sum_k^n \sum_l^n P_1^k P_2^l - \sum_k^n P_1^k \sum_l^n P_2^l)}{n \sum_k^n (P_1^k)^2 - (\sum_k^n P_1^k)^2 + n \sum_l^n (P_2^l)^2 - (\sum_l^n P_2^l)^2}$$



# Κυκλώματα

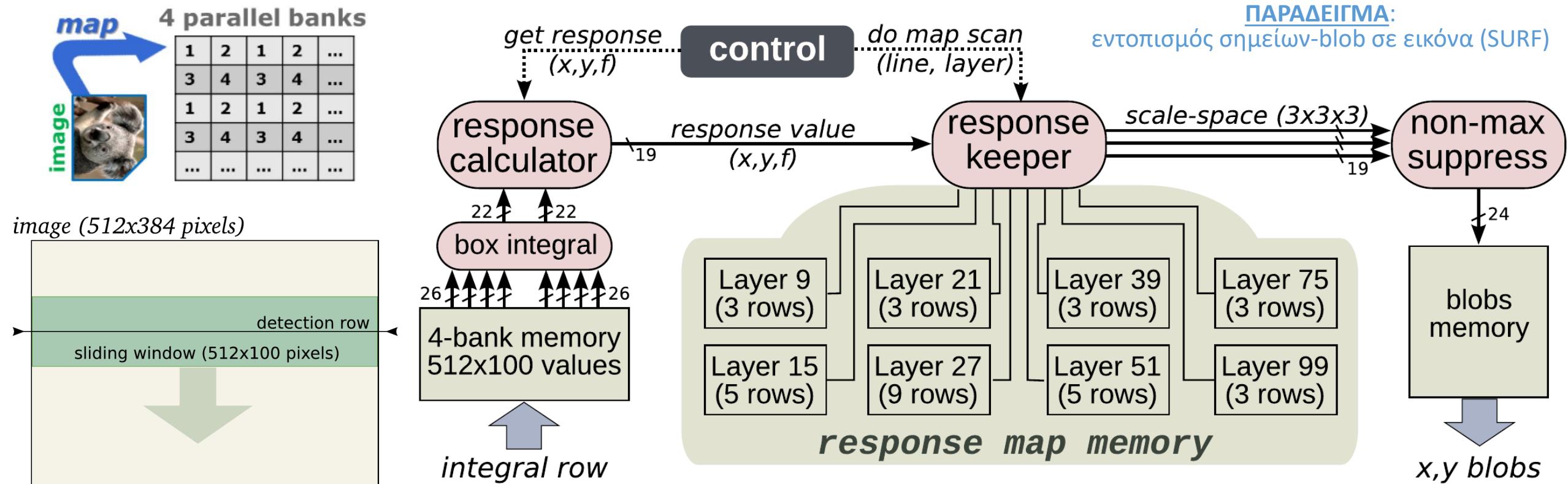
- ξεχωριστές ενδιάμεσες μνήμες: **τα δεδομένα ρέουν** από τη μια στην άλλη μέσω επεξεργασίας
  - λίγο ανισόρροπο pipeline
  - τεράστιο βάθος ( $>1000$ )
- serial-to-parallel converter
  - ροή εισόδου 1--1, αλλά ροή εξόδου  $5 \times 5$ -- $5 \times 5$
  - παραλληλισμός συνέλιξης
- επανάχρηση μονάδων
  - αλλά έξτρα τοπικές μνήμες



**ΠΑΡΑΔΕΙΓΜΑ:**  
εντοπισμός σημείων-γωνιών σε εικόνα (Harris)

# Κυκλώματα

- προσεκτική χρήση περιορισμένης on-chip RAM (sliding window, stripes,...)
- αντιστοίχιση εικόνας σε τράπεζες μνήμης για παράλληλη ανάγνωση pixel

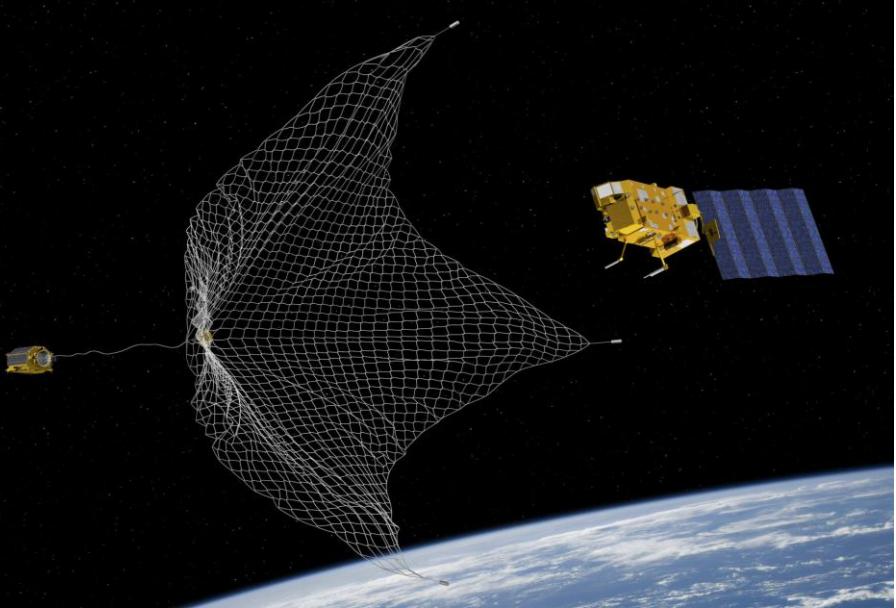


# Εφαρμογή: Διαστημικά Σκουπίδια

- Vision-Based Navigation, αυτόματα με κάμερες+αλγόριθμο
  - για ραντεβού (ADR, servicing, docking) ή προσεδάφιση (landers)

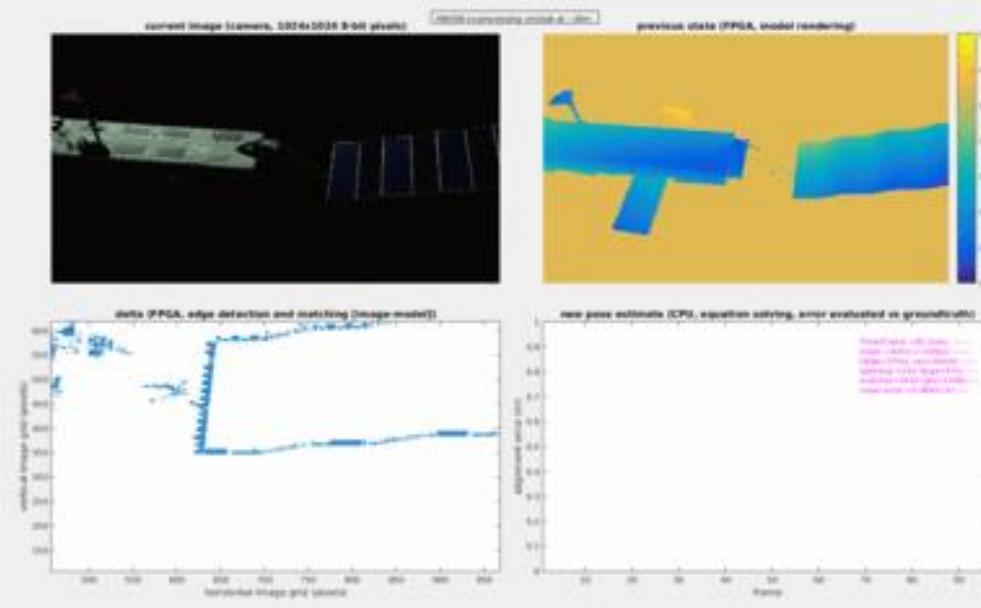
ADR: εκτροχιασμός ανεξέλεγκτων δορυφ.

- με "κυνηγό" και δίχτυ, δαγκάνα, δόρυ
- αλλιώς, φ. Kessler (σαν χιονοστιβάδα)

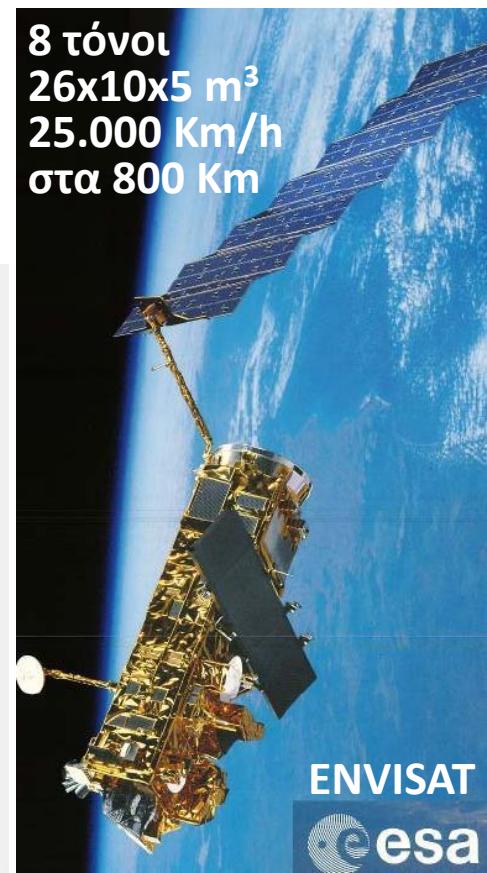


παράδειγμα προσέγγιση ENVISAT (προσομοίωση)

- πάρα πολύ προσεκτικά, με ταίριασμα ορμής
- απαιτήσεις VBN, πχ: 5-10 FPS, 1% σφάλμα



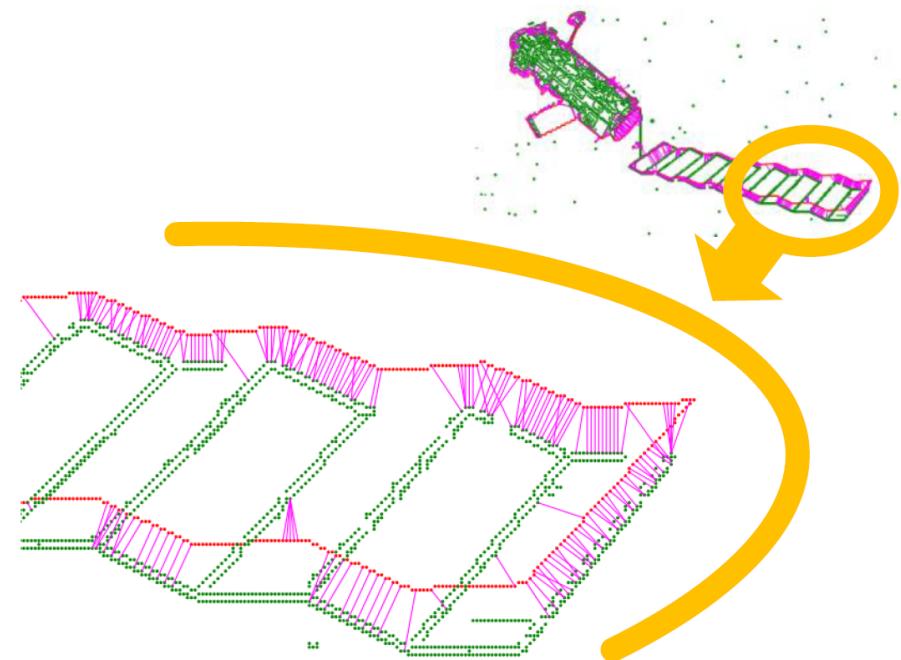
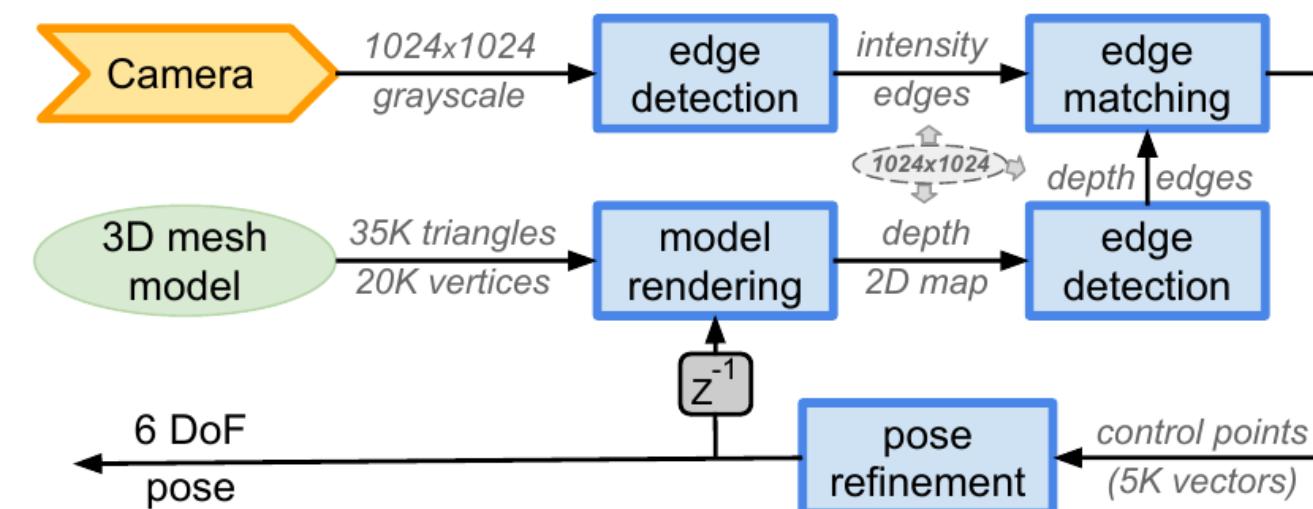
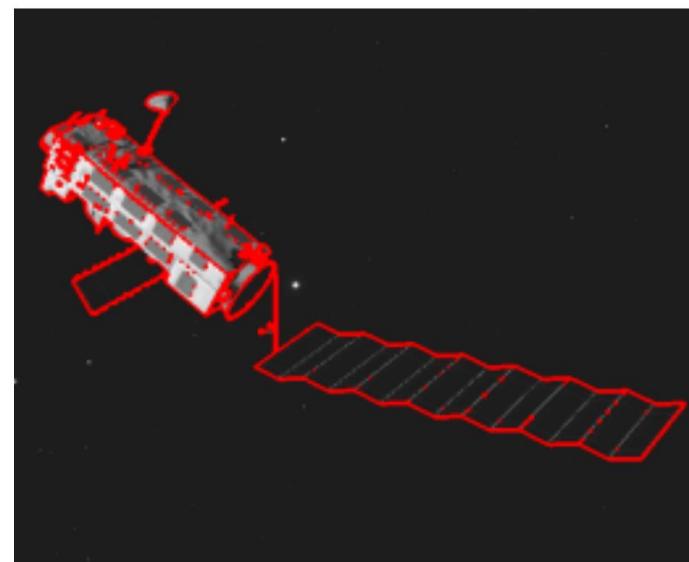
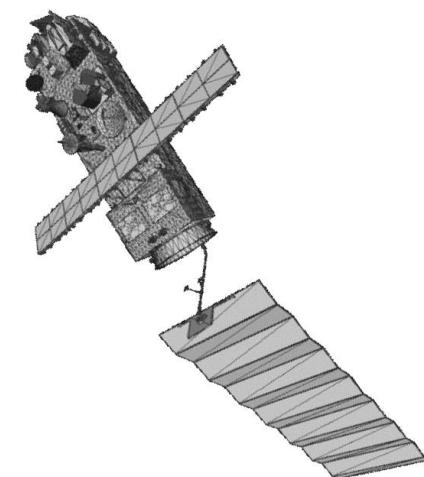
**8 τόνοι  
26x10x5 m<sup>3</sup>  
25.000 Km/h  
στα 800 Km**



ENVISAT  
 esa

# Αλγόριθμοι

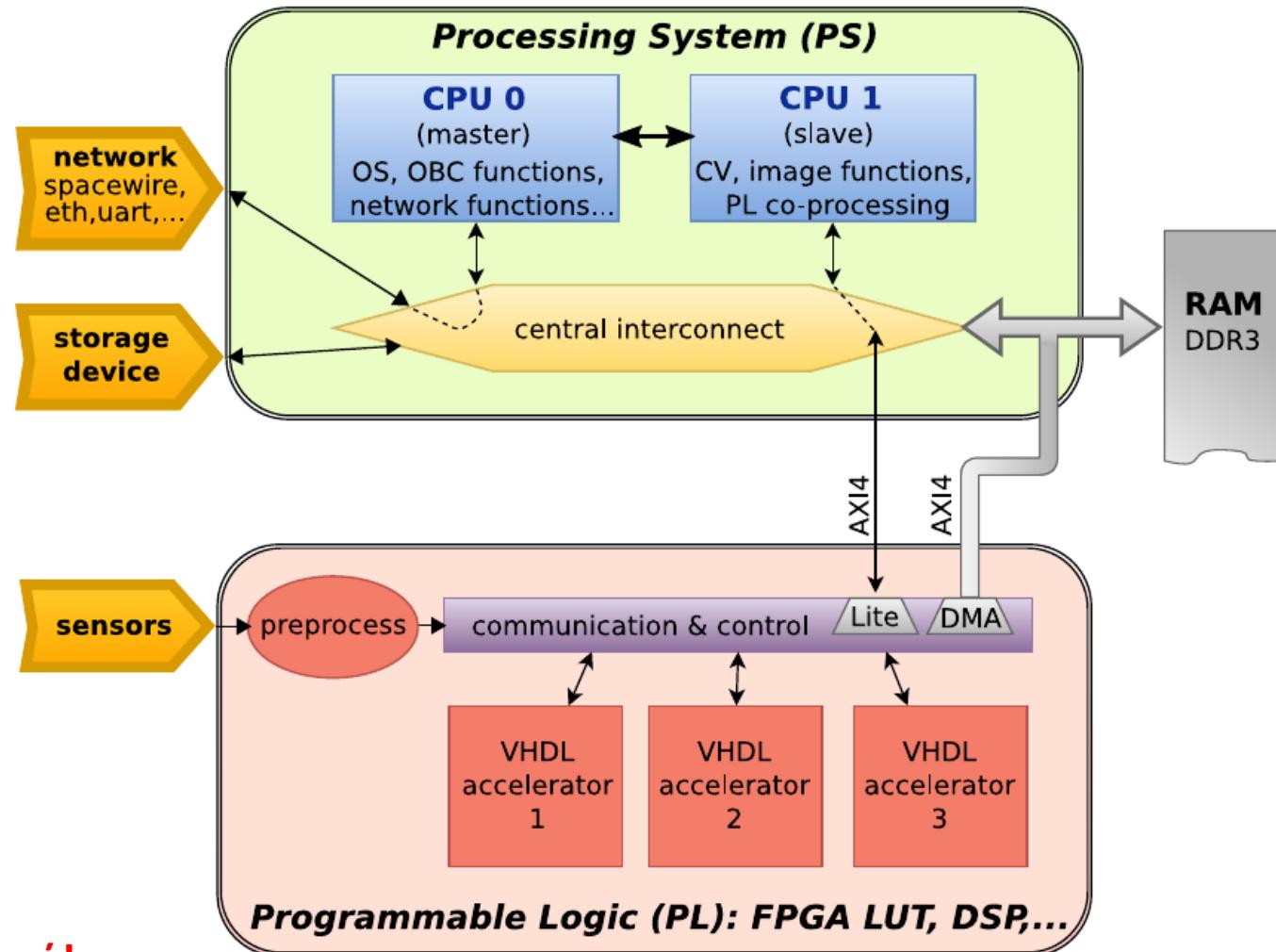
- εντοπισμός και ταίριασμα σημείων-ακμών
  - μεταξύ τρέχουσας εισόδου και εκτιμώμενης κατάστασης (ανανέωση ανά εικόνα, **tracking**)
- rendering 3D μοντέλου με βάση 6D πόζα, Canny edges → σημεία ελέγχου για PoseEst.



# FPGA SoC: HW/SW partitioning & co-processing

- πρώτα μελέτη πολυπλοκότητας

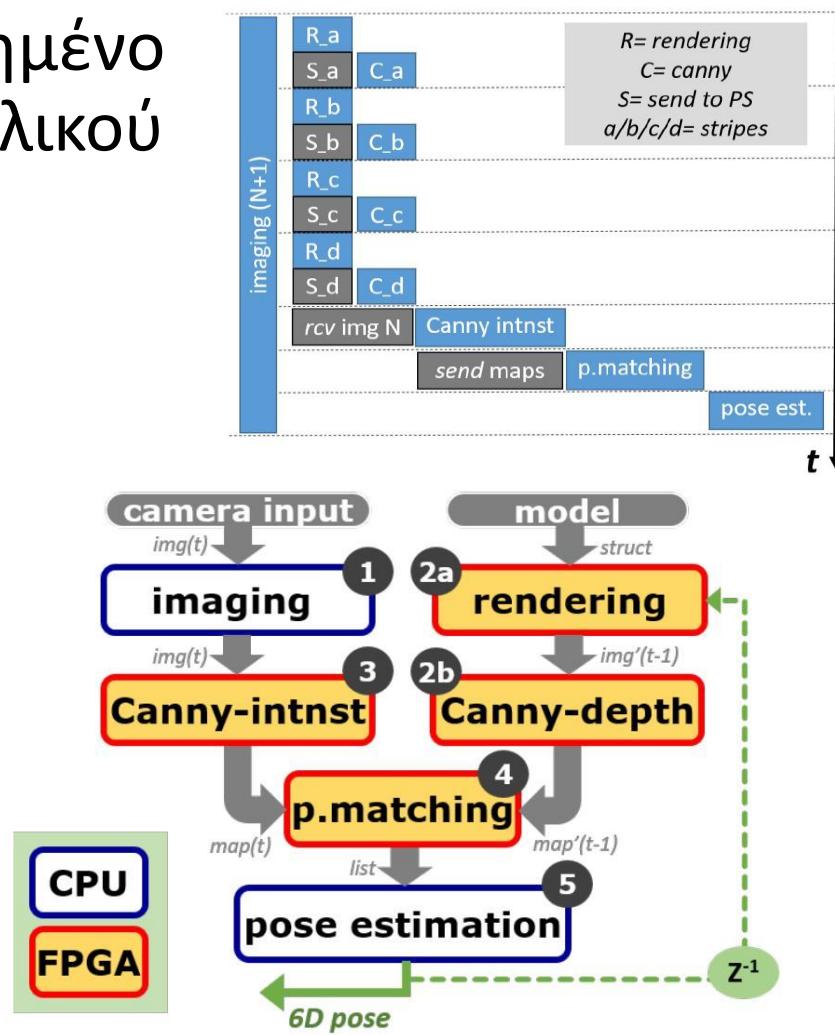
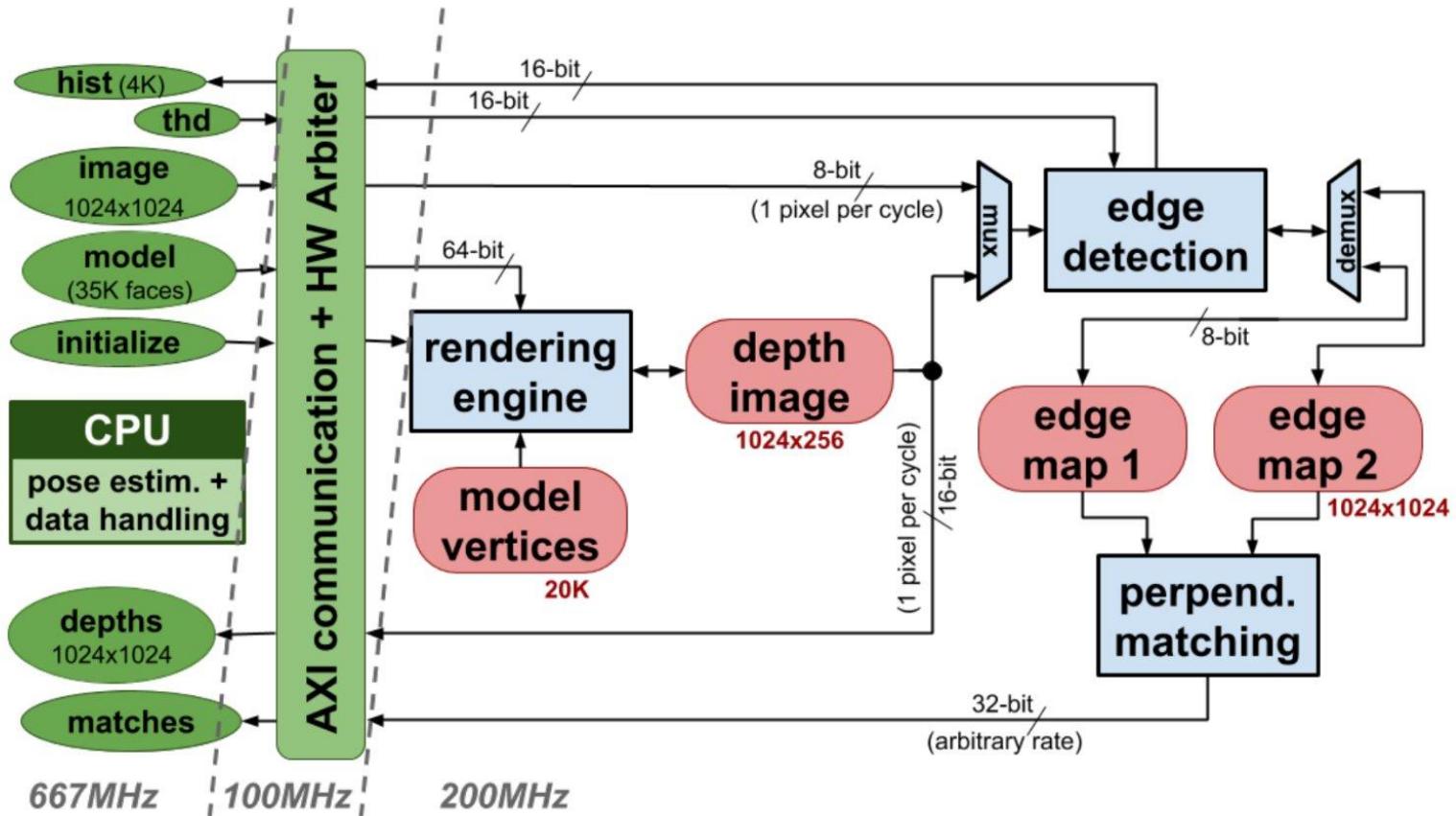
algorithm –function	time / frame (cam@50m)	time / frame (cam@30m)	I/O (Mbit)	RAM (MB)	other issue*
imaging	82 msec	82 msec	16.8	1	
–fetch	56 msec	56 msec	16.8		FX, SP
–enhance	26 msec	26 msec	16.8		
rendering	343 msec	869 msec	52	7	
–initialize	0.1 msec	0.1 msec	0.01		
–projtrian	7.5 msec	10 msec	125		
–gen+chk	206 msec	544 msec	4211		
–pixdepth	51 msec	153 msec	659		
–other	78.4 msec	162 msec	$\leq 1$		
edgedetect	303 msec	326 msec	12.6	18	
–gradient	210 msec	210 msec	41.9		HP+SP,
–suppress	44 msec	44 msec	67.1		FX
–hysterss	16 msec	29 msec	67.1		
–other	33 msec	43 msec	33.5		
matching	38 msec	39 msec	17.6	2.2	SP, FX
pose estim	51 msec	99 msec	$\leq 1.6$	$\leq 1$	
–LQS-fit	16 msec	32 msec	$\leq 1.6$		PC, EM,
–LSinlier	33 msec	65 msec	$\leq 1.6$		FL
–other	2 msec	2 msec	$\leq 1.6$		
<i>total**:</i>	1120 msec	1741 msec	8.4	44	–



δεν είναι μόνο ο χρόνος εκτέλεσης το κριτήριο διαχωρισμού!

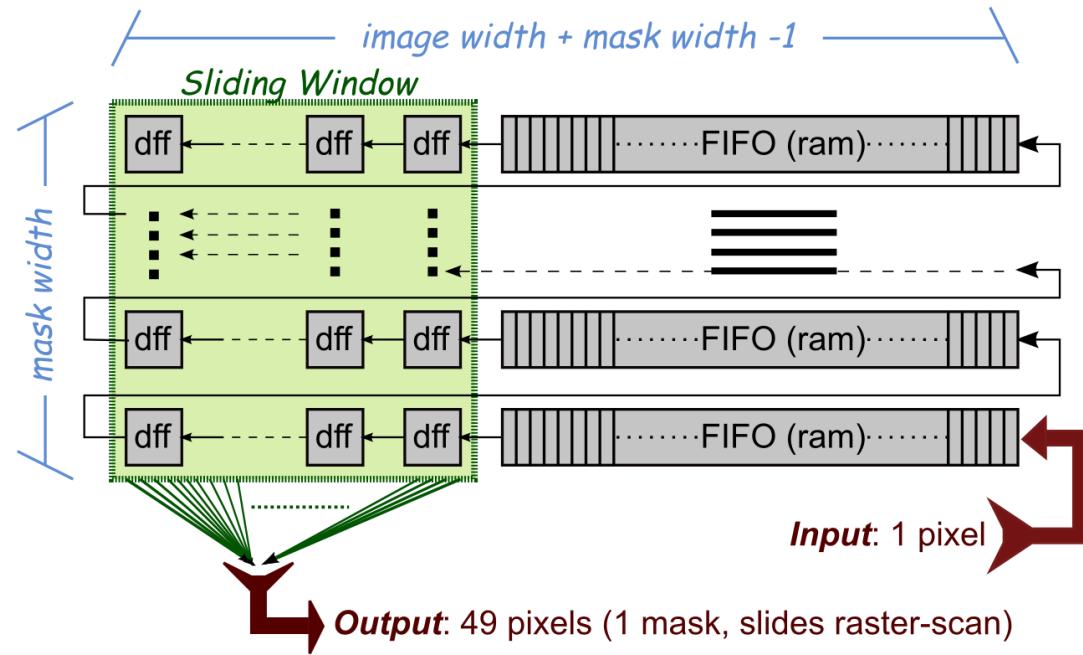
# Αρχιτεκτονική, αντιστοίχιση & χρονοπρόγραμμα

- διαφορετικά πεδία (CPU/comm/FPGA), ισορροπημένο μοίρασμα φόρτου, για μέγιστη εκμετάλλευση υλικού

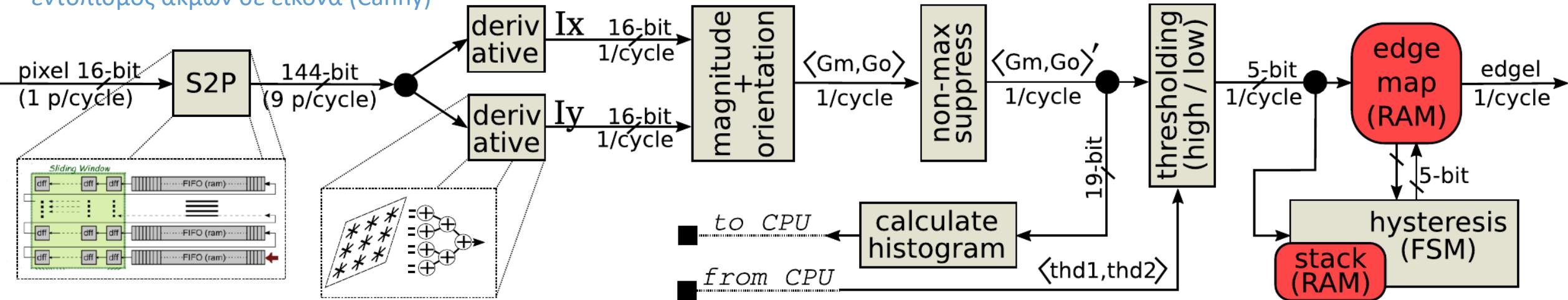


# Κυκλώματα

- 2D sliding window (S2P, 1 σε 49)
  - σωστό μίγμα RAMB και DFF ( $7 + 7 \times 7$ )
  - τεράστιο pipe από είσοδο έως μνήμη
    - όλες οι πράξεις σε 1 διάβασμα εικόνας!
  - μεταβαλλόμενη αριθμητική (16-5 bit)



ΠΑΡΑΔΕΙΓΜΑ:  
εντοπισμός ακμών σε εικόνα (Canny)



# Αποτελέσματα - Σχόλια

- 1-3 τάξεις μεγέθους γρηγορότερη εκτέλεση σε σχέση με σκέτη μικρή CPU
  - 2-3 σε επίπεδο συνάρτησης (πχ, 3 για stereo)
    - σημ.: RH-CPU είναι 2-10x πίσω από συνηθισμένα
  - 1 σε επίπεδο πλήρους συστήματος HW/SW
- με την ακρίβεια να παραμένει σχεδόν ίδια
- και την ισχύ σε ανεκτά πλαίσια 1-10 Watt
- επιταχύνουμε το 80-99% υπολογισμών
  - προτιμάμε το "number crunching" για FPGA
  - και αφήνουμε στη CPU μικρές κι ακριβές συναρτήσεις που καλούνται περιστασιακά
    - γιατί στο HW υπάρχουν/μένουν όλη την ώρα
    - ή άλλες που θέλουμε να πειράζουμε εύκολα



vs LEON3 (150MIPS)

Spacesweep = 637x

Disparity = 120x

Harris = 75x

SURFdet = 56x

SURFdesc = 84x

SIFTdesc = 100x

SIFTmatch = 180x

BRIEFmatch = 100x

SYSTEM = 20x

(or 444x for stereo)

vs ARM-A9 (667mhz)

Canny = 55x

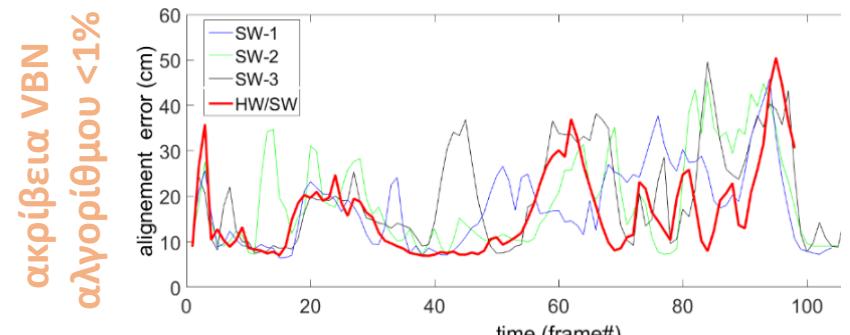
Rendering = 62x

SYSTEM = 19x

(or 10x vs LEON4)

VBN thrput = 10FPS

Localization = 1 FPS



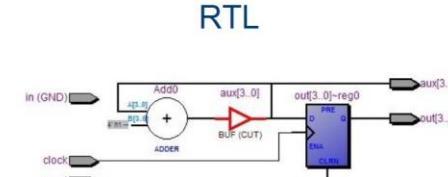
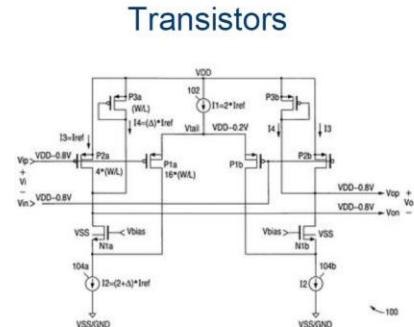
# HLS

High Level Synthesis:  
προγραμματισμός FPGA / σχεδίαση ASIC, χωρίς γλώσσες HDL

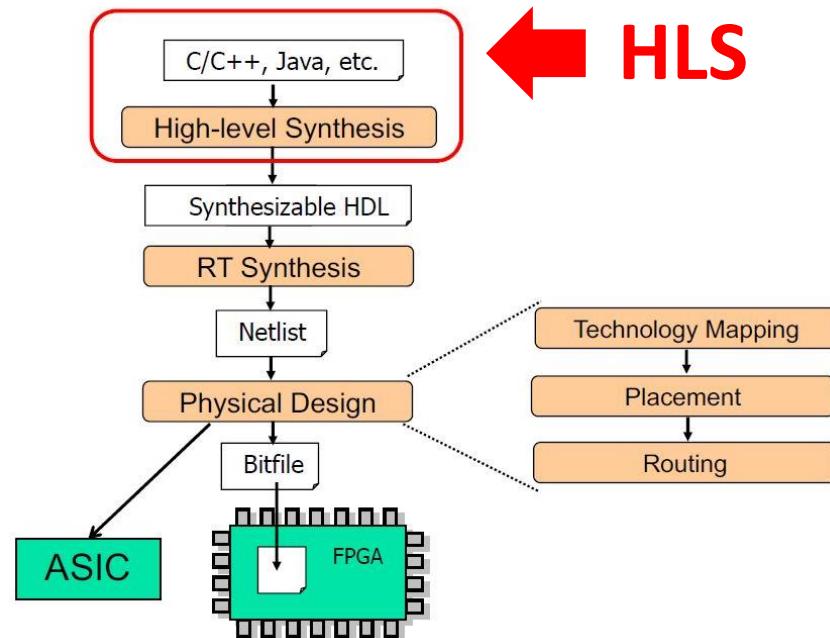
- XILINX
  - <https://docs.amd.com/v/u/en-US/ug998-vivado-intro-fpga-design-hls>
  - <https://docs.amd.com/r/2022.1-English/ug1399-vitis-hls/Basics-of-High-Level-Synthesis>
  - [https://www.xilinx.com/support/documents/sw\\_manuals/xilinx2022\\_2/ug1399-vitis-hls.pdf](https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_2/ug1399-vitis-hls.pdf)
- CADENCE
  - <https://www.slideshare.net/slideshow/highlevel-synthesis-for-the-design-of-ai-chips/266985270>
- Other
  - Lahti, Sakari, and Timo D. Hämäläinen. "High-level Synthesis for FPGAs-A Hardware Engineer's Perspective." IEEE Access, vol.13, 10.1109/ACCESS.2025.3540320 (2025)
  - High-Level Synthesis Creating Custom Circuits from High-Level Code, Hao Zheng ,Comp Sci & Eng, University of South Florida

# Τι είναι HLS ?

- αντί για HDL/RTL, περιγράφουμε το κύκλωμα με υψηλή γλώσσα, πχ, με C/C++ και directives/pragmas
- **επιχειρήματα υπέρ** (ευκολία ανάπτυξης)
  - πιο γρήγορη ανάπτυξη και αλλαγή κώδικα
  - μεγαλύτερο κοινό/διαθεσιμότητα μηχανικών
- **επιχειρήματα κατά** (ποιότητα κυκλώματος)
  - η αφαίρεση δεν επιτρέπει λεπτομερή σχεδίαση
  - έλλειψη γνώσης HW δεν επιτρέπει καλή σχεδίαση
- HLS προσπάθεια δεκαετιών, μεγάλη δυσκολία



Abstraction and Productivity



# Σημερινή Εικόνα HLS

## • βιομηχανικά εργαλεία

- Vitis/Vivado HLS (AMD Xilinx) [FPGA]
- Intel oneAPI HLS (Intel Altera) [FPGA]
- SmartHLS (Microchip) [FPGA]
- Catapult C (Siemens Mentor) [+ASIC]
- Stratus HLS (Cadence) [+ASIC]
- ...

## • ακαδημαϊκά/ανοιχτά

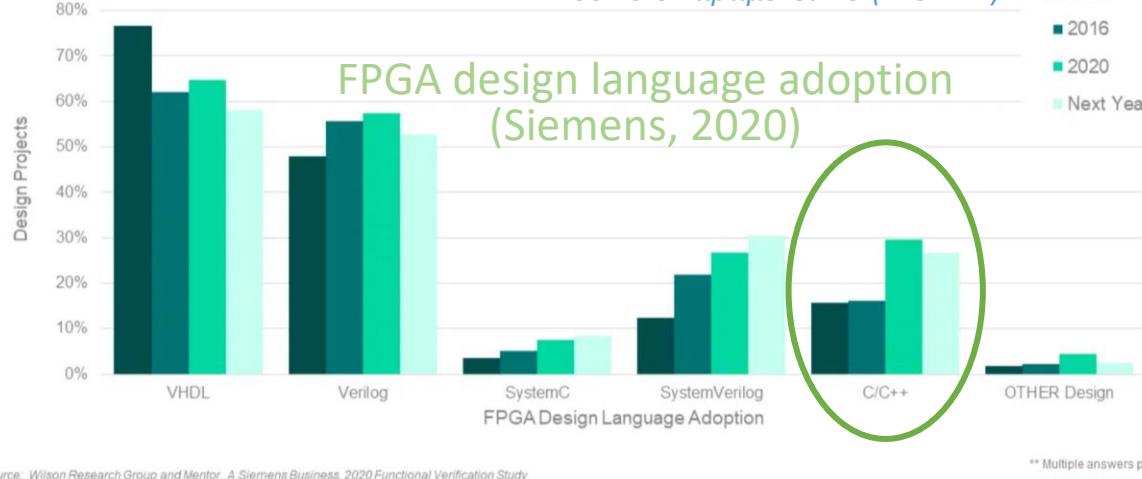
- PandA-Bambu (Milan), LegUp (Toronto)
- HLS4ML, OpenHLS
- ...

- HLS πλέον αρκετά ώριμο, πιο ευρεία χρήση
  - 90's έρευνα, 00's Catapult, 10's FPGA, 20's AI...

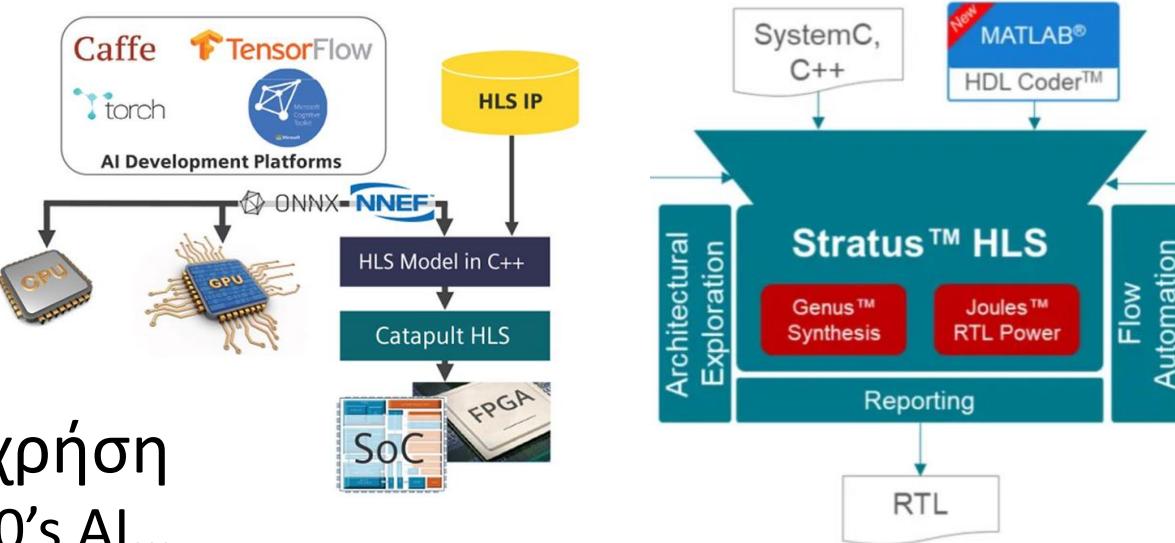
Year	Papers	Applications
2017	7	15
2018	8	14
2019	5	8
2020	12	20
2021	6	12
2022	7	10
2023	5	11
2024	1	3
<b>Total</b>	<b>51</b>	<b>93</b>

είπαν...

- "εργαλεία HLS≈1B\$ της αγοράς!"
- θέβαια τα RTL≈10B\$
- "σε FPGA τα κυκλώματα είναι 1:3"
- σε ASIC παραμένει 1:9 (HLS:HDL)

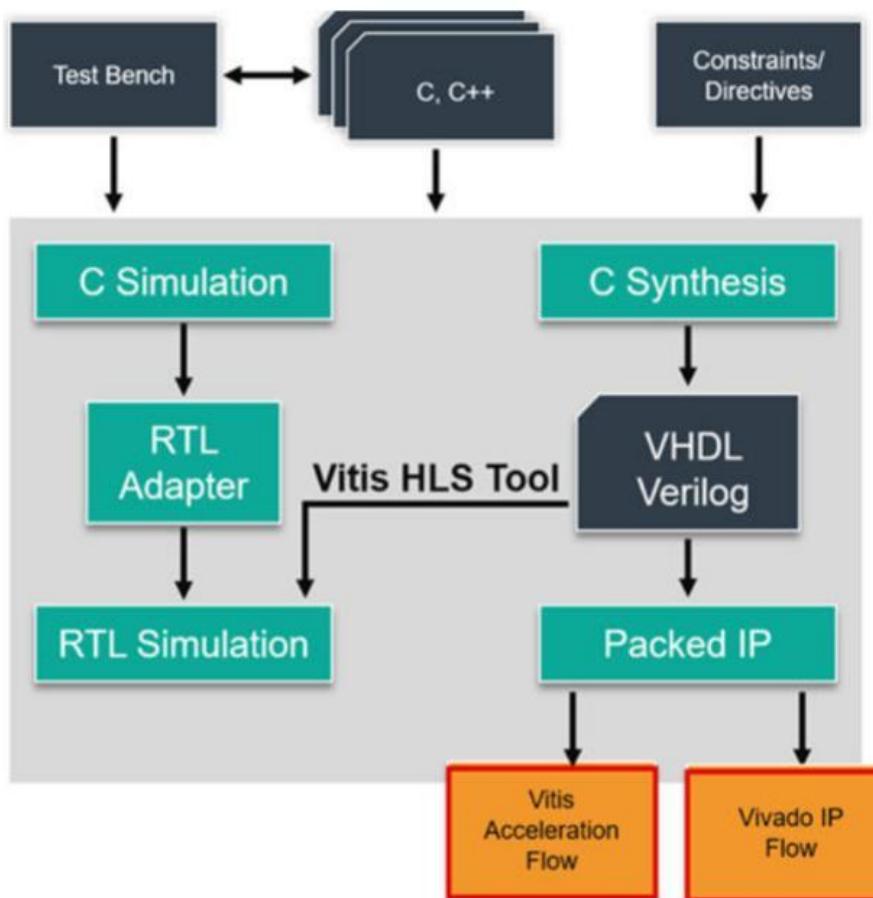


Source: Wilson Research Group and Mentor, A Siemens Business, 2020 Functional Verification Study  
Page 1 © Siemens 2020 | 2020-10-15 | Siemens Digital Industries Software | Where today meets tomorrow.



# AMD/Xilinx HLS

- εργαλείο μέσα στο Vivado/Vitis: είσοδος C++ και constraints, έξοδος HDL



1. Write the algorithm at a high abstraction level using C/C++/SystemC with a given architecture in mind
2. Verify the functionality at the behavioral level
3. Use the HLS tool to generate the RTL for a given clock speed, input constraints
4. Verify the functionality of the generated RTL
5. Explore different architectures using the same input source code

τυπική ακολουθία βημάτων ανάπτυξης κυκλωμάτων (επάνω) και διαχωρισμός ευθυνών σχεδιαστή-εργαλείου (δεξιά) [σύμφωνα με Xilinx]  
Ιδιαίτερη σημασία στην εξερεύνηση λύσεων (5)

Macro Architecture  
Design Intent  
Constraints  
  
FSM Generation  
Schedule of Operations  
Clock  
Pipelining Registers  
Sharing Resources  
Timing  
Verification

Designer's Responsibility  
  
Automation by HLS Tool

# Από C/C++ σε HDL

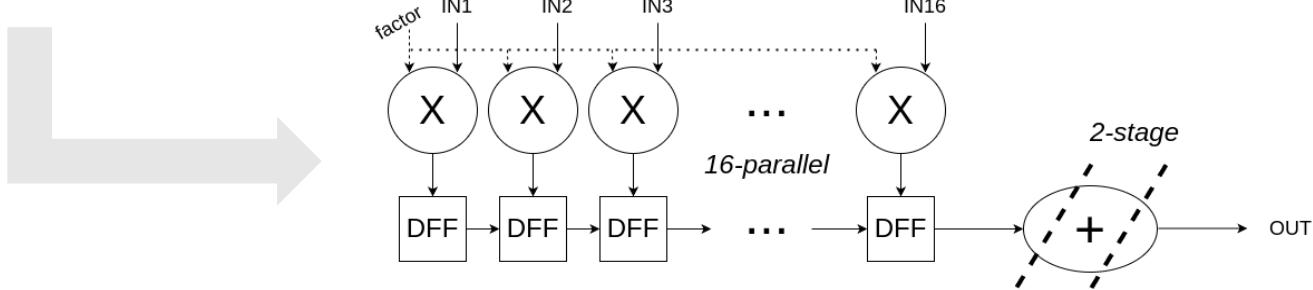
- βασίζεται σε 2 παρατηρήσεις
  - συνήθεις δομές SW αντιστοιχούν σε συνήθεις δομές HW
  - οδηγίες σε compiler μπορούν να κατευθύνουν σε βελτιστοποιήσεις ή/και trade-offs

Σημ.1: το δυσκολότερο στοιχείο που πρέπει να εξαχθεί από το SW είναι ο παραλληλισμός. Για πιο συνήθεις δομές χρειάζεται καλό κώδικα (πχ, χωρίς εξαρτήσεις) και σωστές οδηγίες. Στην πιο γενική περίπτωση, είναι ανοιχτό ερώτημα θεωρητικής πληροφορικής ("P vs NC", εφάμιλλο του "P vs NP")

Σημ.2: μικρές αλλαγές στο SW μπορεί να επιφέρουν μεγάλες αλλαγές στο HW (βλ. κι "εξερεύνηση"). Επίσης, χρειάζεται πάντα επαλήθευση στο παραγόμενο κύκλωμα HLS (σωστό SW δεν εγγυάται πάντα σωστό HW)

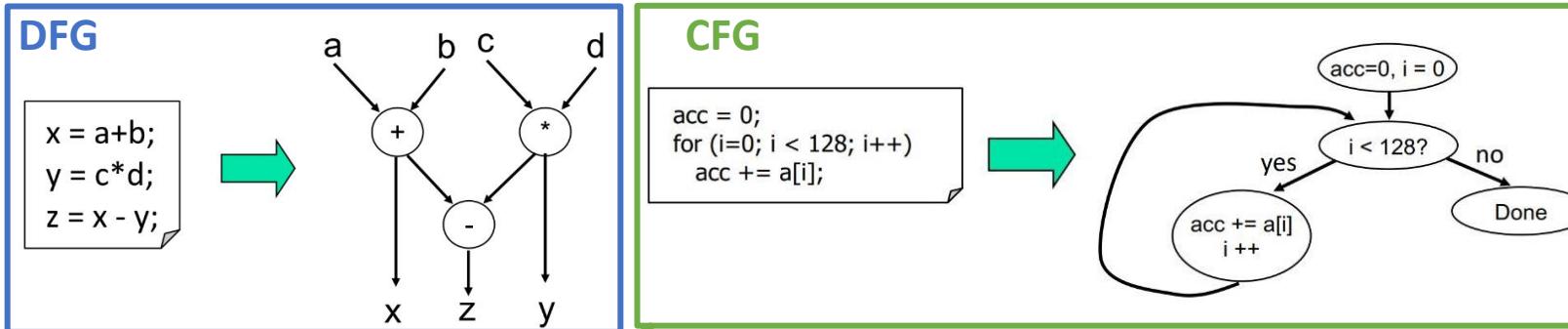
```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #define N 16
4
5 // Component 1: Fully parallel array scaling function
6 void array_scale_parallel(int in[N], int out[N], int factor) {
7     #pragma HLS ARRAY_PARTITION variable=out complete dim=1
8     scale_loop: for (int i = 0; i < N; i++) {
9         #pragma HLS UNROLL
10        out[i] = in[i] * factor;
11    }
12
13 // Component 2: Pipelined array increment function
14 void array_increment_pipelined(int in[N], int out[N]) {
15     increment_loop: for (int i = 0; i < N; i++) {
16         #pragma HLS PIPELINE II=1
17         #pragma HLS RESOURCE variable=out core=AddSub_DSP latency=2
18        out[i] = in[i] + 1;
19    }
20
21 // Top-level function: chained data flow
22 void dflow(int in[N], int aux[N], int out[N], int factor) {
23     array_scale_parallel(in, aux, factor);
24     array_increment_pipelined(aux, out);
25 }
```

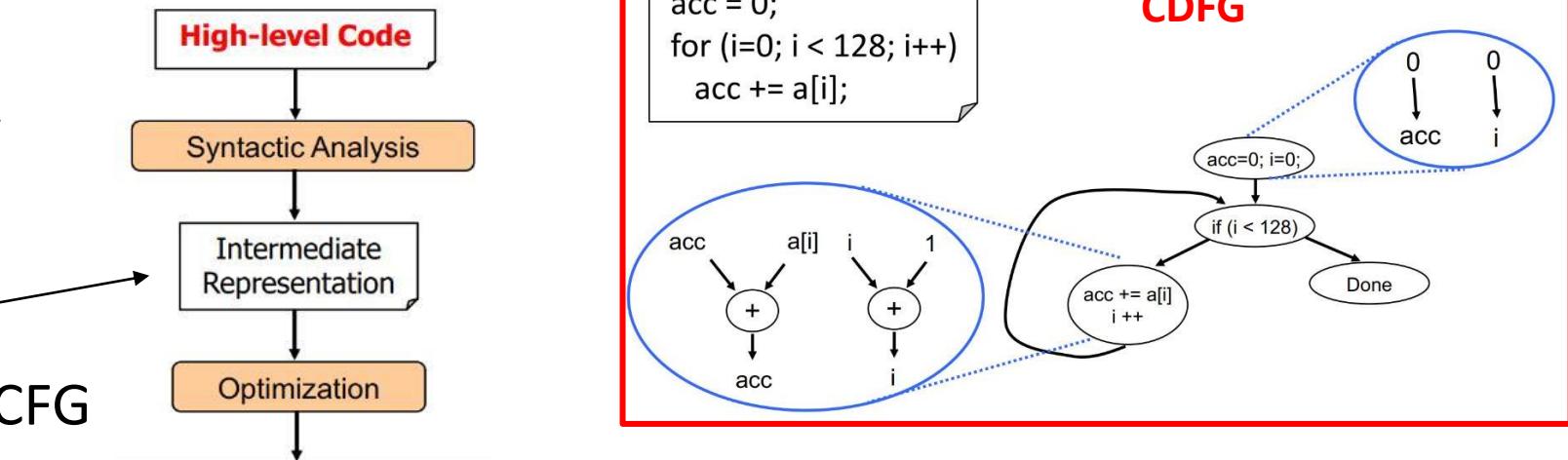


# Flow Graphs (ενδιάμεση αναπαράσταση IR)

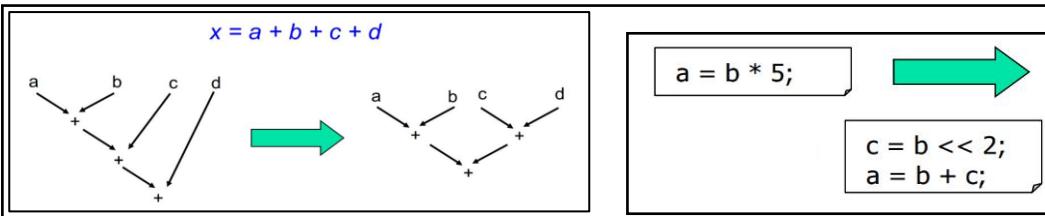
- Data Flow Graph
  - για εξαρτήσεις data εντός βασικού μπλοκ
  - σκέψου **datapath**



- Control Flow Graph
  - μεταξύ βασικών μπλοκ
  - σκέψου **control path**

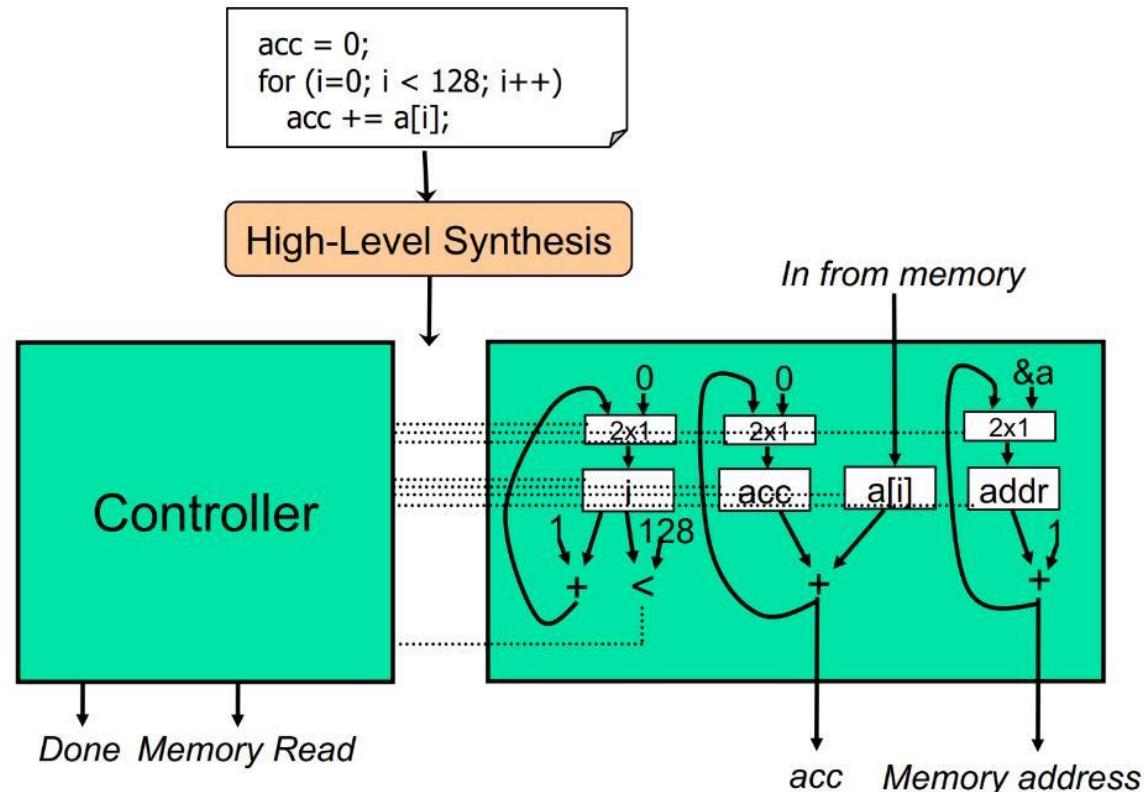
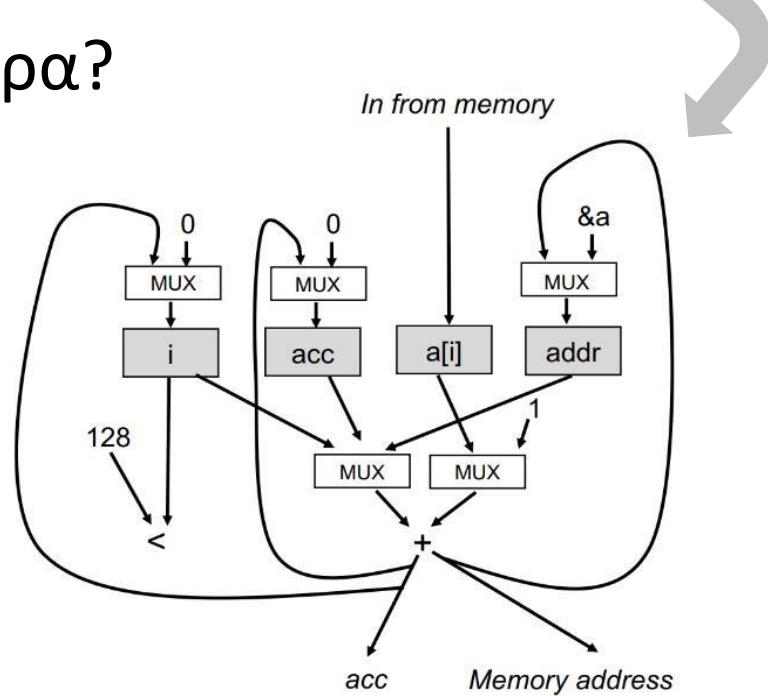


- συνδυασμός = **CDFG**
  - 1 DFG για κάθε κόμβο CFG
  - από κώδικα-είσοδο προς κύκλωμα-έξοδο, δημιουργείται αυτόματα
  - βελτιώνεται (ύψος δέντρου, σταθερές,...)



# παράδειγμα 1α

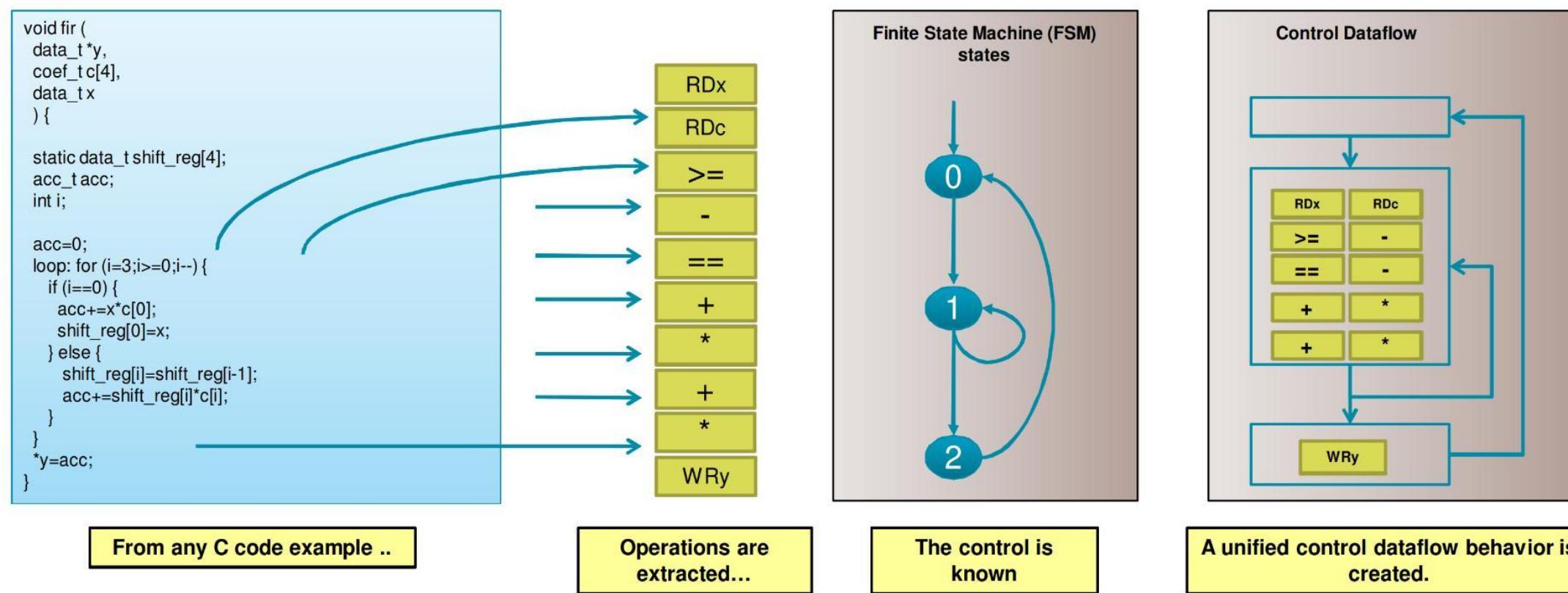
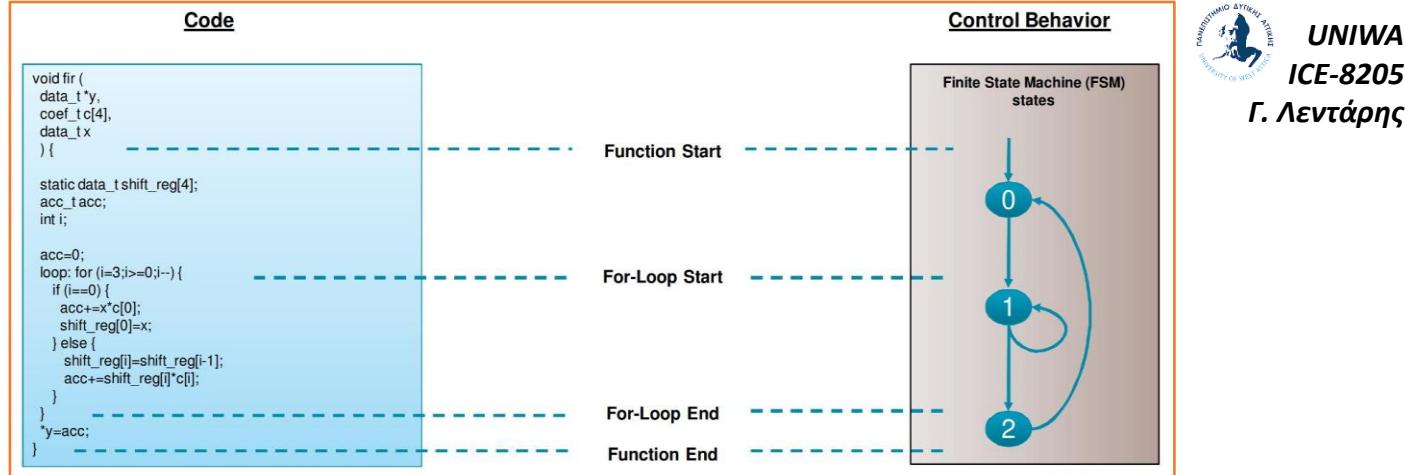
- απλός βρόχος συσσώρευσης
  - φτιάχνει FSM για controller
  - αθροιιστές για acc και διευθύνσεις
- πιο οικονομικά: επανάχρηση "+"
- πιο γρήγορα?



**Χρειάζεται:**  
 1) scheduling  
 2) resource allocation  
 3) binding (mapping)

# παράδειγμα 1β

- FIR: πιο πολύπλοκο loop...

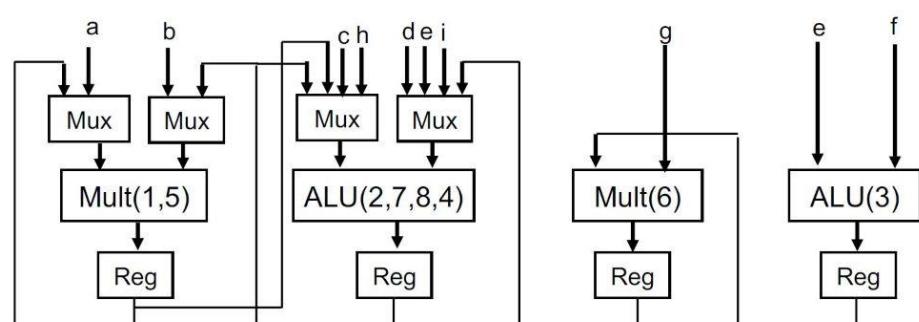
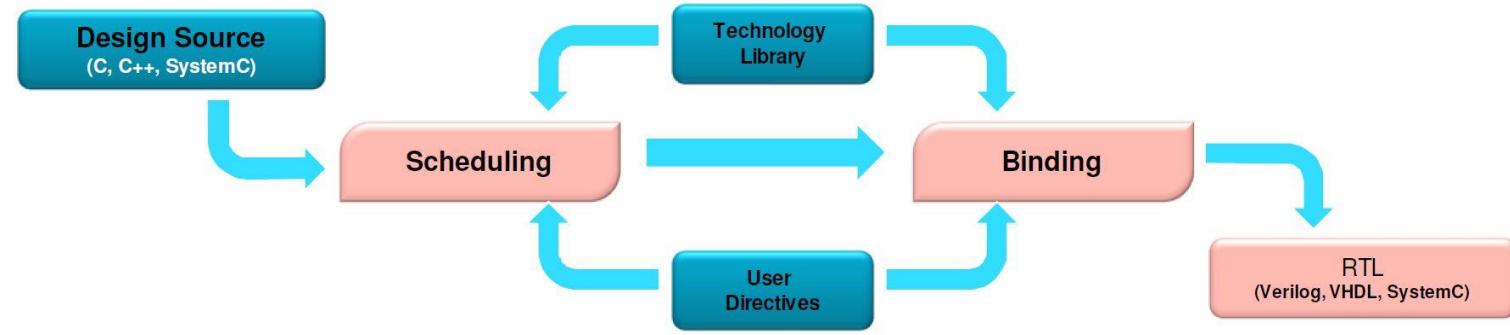
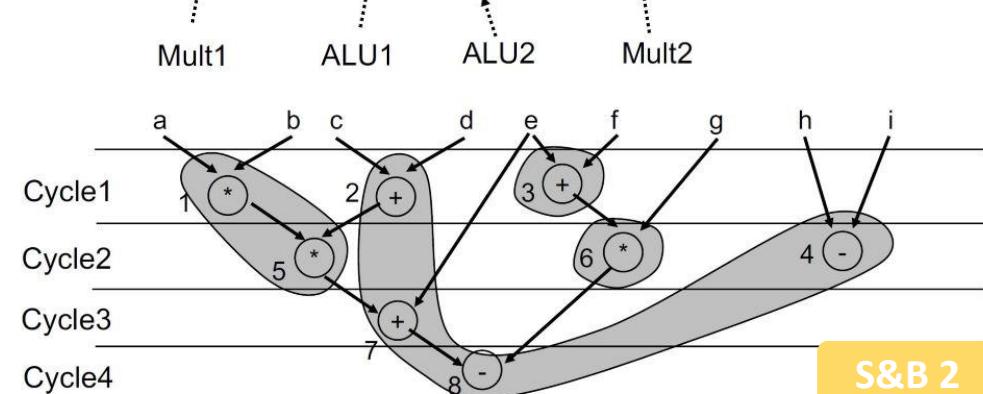
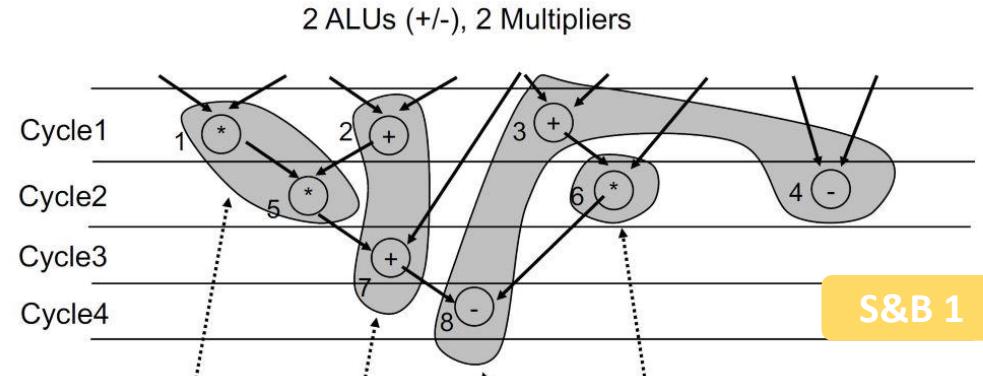


# Scheduling & Binding

- σε ποιο κύκλο εκτελείται κάθε πράξη
- και σε ποια φυσική μονάδα/θέση HW
- ρόλο: κώδικας, περιορισμοί, τεχνολογία
  - καθυστερήσεις στοιχείων, στόχος χρήστη
  - εξαρτήσεις, πόσες πράξεις στον κύκλο ( $f_{clk}$ )
  - και διαθεσιμότητα/οικονομία κυκλωμάτων

πχ, απλές πράξεις: επανάχρηση πόρων, διαφορετικά S&B (με άλλη απόδοση?), οδηγούν στο τελικό κύκλωμα

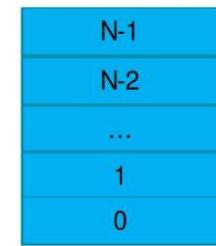
$$x = [(a*b)*(c+d)+e] - [(e+f)*g], \quad y = h - i$$



# Πίνακες (μνήμες)

- Vitis HLS "μεταφράζει" C array σε μνήμη
  - με FIFO, RAMB, LUTRAM (default=single-port)
  - προτιμάει static, απαγορεύεται dynamic!
  - επιτρέπει αρχικοποίηση (*int coeff[8] = {-2, 8, -4,...}*)
  - προσοχή στο μέγεθος (να χωράει on-chip)
  - σημ: όχι πάντα ίδια 'μετάφραση' ο άνθρωπος...
- αν ο πίνακας είναι στο όρισμα της *top-level* συνάρτησης υποθέτει off-chip RAM
  - βάζει m-axi port interface με latency=1
  - θέλει οδηγίες/πρωτόκολλο (βλ. παρακάτω)
- μπορεί να δημιουργήσουν bottlenecks!
  - θέλει οδηγίες/βελτιστοποίηση (βλ. παρακάτω)

```
void foo_top(int x, ...)  
{  
    int A[N];  
    L1: for (i = 0; i < N; i++)  
        A[i+x] = A[i] + i;  
}
```



A[N]

Synthesis



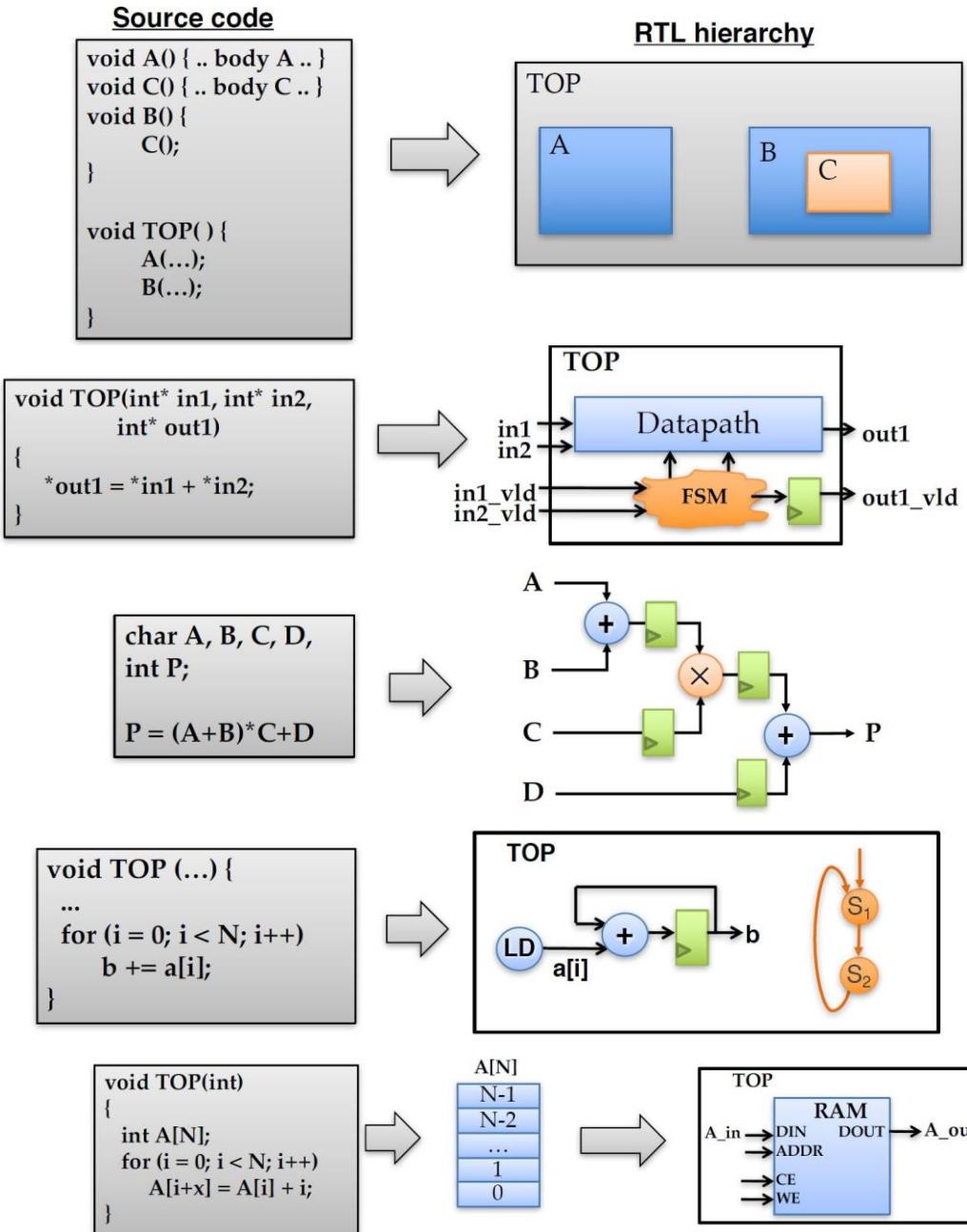
```
1 dout_t array_mem_bottleneck(din_t mem[N]) {  
2  
3     dout_t sum=0;  
4     int i;  
5  
6     SUM_LOOP:for(i=2;i<N;++i)  
7         sum += mem[i] + mem[i-1] + mem[i-2];  
8  
9     return sum;  
10 }
```

mem[N] --> εξωτερική RAM

# Συνήθεις δομές, συνοπτικά

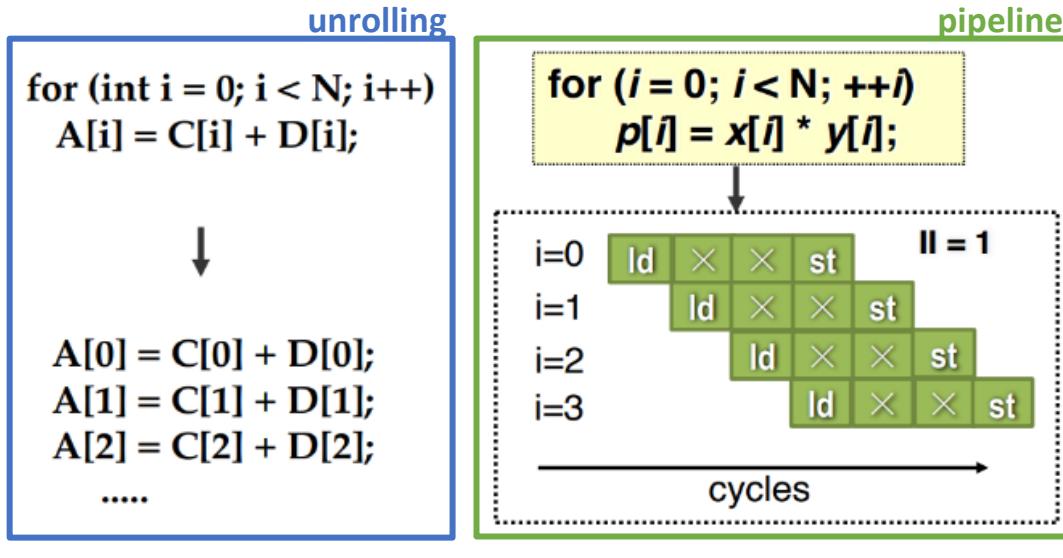
- HLS “ανιχνεύει” κι αντιστοιχεί σε HW
  - ως μια βάση, προς εξειδίκευση με οδηγίες

software (C/C++)	hardware (RTL)
functions	modules/components
arguments	I/O ports (with protocols)
math expressions/ops	datapath/ALU logic
conditionals (if,...)	mux/control logic
loops	pipeline & FSM
arrays	memories
scalar variables	wires, registers
structs	bundled wires, memories
simple pointers	address generators
queuesstreams	FIFOs
...	...



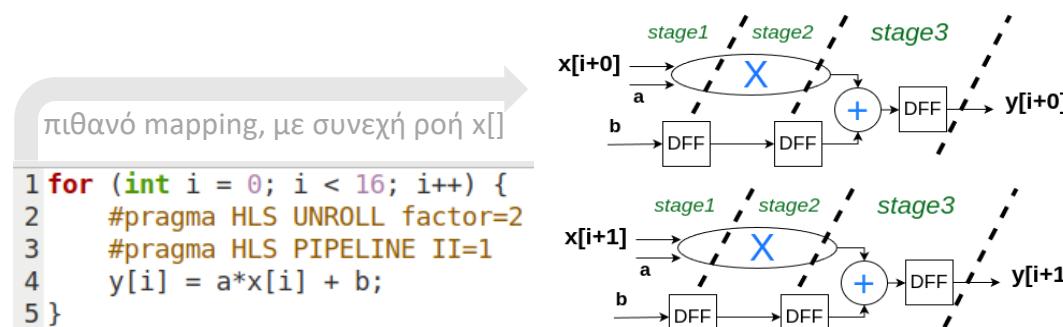
# HLS pragmas: loop unroll / pipeline

- “default loops are rolled” ⇒ σειριακά!
- **οδηγία unrolling ⇒ παραλληλισμός**
  - μέσα, σαν concurrent statements (VHDL)
  - το loop πρέπει να έχει στατικό μήκος
  - ορίζω unroll factor (παράλληλες πράξεις)
  - **κόστος αυξάνεται** (area++, power++,  $f_{clk}-$ )
  - για να αποδώσει, πρέπει και τα data να διαβαστ/γραφτούν παράλληλα στις RAM
- **οδηγία pipeline ⇒ παραλληλισμός**
  - βασική παράμετρος: *Initiation Interval* (II)
    - κάθε πόσους κύκλους εισάγει νέο δεδομένο
    - II=1 ⇒ fully pipelined, II=2/3/... ⇒ HW reuse
  - μπορεί να συνδυαστεί με partial unrolling



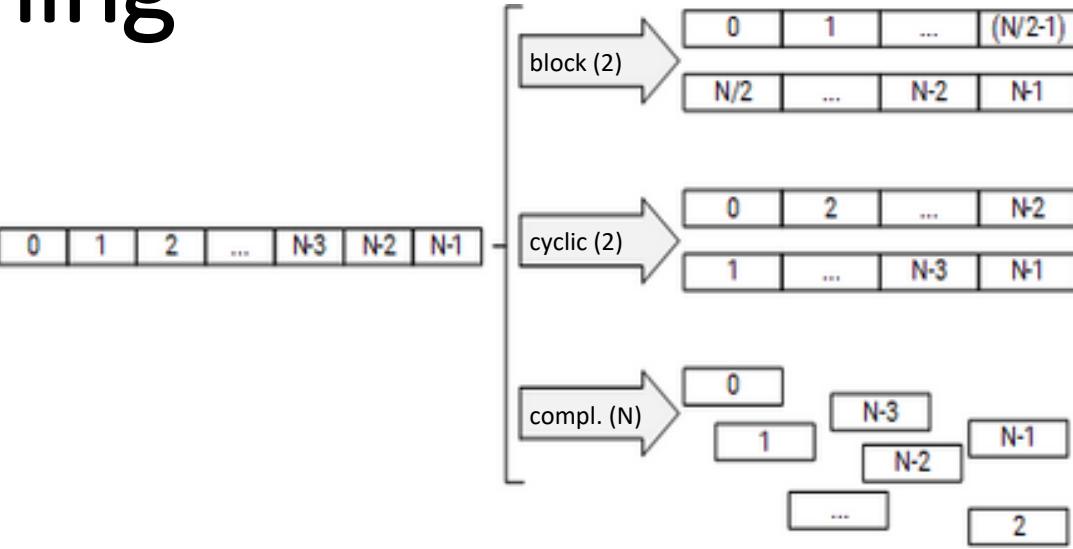
```
1 loop: for (int i = 0; i < N; i++) {
2   #pragma HLS UNROLL factor=3
3   a[i] = c[i] + d[i];
4 }
```

```
1 loop: for(int i = 0; i < 5; i++) {
2   #pragma HLS PIPELINE II=1
3   a[i] = b[i] + c[i];
4 }
```



# HLS pragmas: array partitioning

- για παράλληλο γράψιμο/διάβασμα δεδομένων στη μνήμη (throughput++)
  - προς κλασικό mem banking/interleaving
    - πολλαπλές μνήμες αντί για 1, με ανεξάρτητη πρόσβαση (προσοχή σε πλήθος & patterns)
- τρεις βασικοί τρόποι, συν παράμετροι
  - *complete*: όλα ξεχωριστά (με registers, κόστος)
  - *block*: cont. chunks ( $B_0=\{1,2,3\dots\}$   $B_1=\{11,12,13\dots\}$ )
  - *cyclic*: interleaved ( $B_0=\{1,11,21\dots\}$   $B_1=\{2,12,22\dots\}$ )
    - factor: σε πόσους υπο-πίνακες να σπάσει
    - dim: προς ποια διάσταση να σπάσει
- καλό να ακολουθεί παράλληλη επεξεργ.
  - πχ, loop unroll ίσου factor, ή πολλά pipe



```
int buf_in[16];
int buf_out[16];

#pragma HLS ARRAY_PARTITION variable=buf_in  cyclic factor=4
#pragma HLS ARRAY_PARTITION variable=buf_out cyclic factor=4

for (int i = 0; i < 16; i++) {
    #pragma HLS UNROLL factor=4
    buf_out[i] = buf_in[i] + a;
}
```

εδώ φτιάχνει 4x banks και  
Θα διαβαστούν παράλληλα

```
int top(SS *a, int b[4][6], SS &c) {
    #pragma HLS array_partition type=complete dim=1 variable=b
    ...μπορεί να μπει και στα I/O (εδώ θεωρεί 4x 32-bit πόρτες)
```

# HLS pragmas, συνοπτικά

- 30 οδηγίες, με παραμέτρους (βλ. AMD manual)
  - +67% στη δεκαετία (*loop\_flatten/merge/tripcnt, perf,...*)
  - μπαίνουν ως constraints (πάντα ελέγχουμε report!)

This example partitions the second dimension of the two-dimensional `in_local` array into individual elements.

```
int in_local[MAX_SIZE][MAX_DIM];
#pragma HLS ARRAY_PARTITION variable=in_local type=complete dim=2
```

Removes reset from variable `a` in function `foo` even when the global reset setting is `state` or `all`.

```
void foo(int in[3], char a, char b, char c, int out[3]) {
    #pragma HLS reset variable=a off
```

In this example, array `B` is set to streaming with a FIFO depth of 12.

```
#pragma HLS STREAM variable=B depth=12 type=fifo
```

Function `foo` is specified to have a minimum latency of 4 and a maximum

```
int foo(char x, char a, char b, char c) {
    #pragma HLS latency min=4 max=8
    char y;
    y = x*a+b+c;
    return y
}
```

In the following example, a two-stage pipelined multiplier using fabric logic is specified to implement the multiplication for variable `c` of the function `foo`.

```
int foo (int a, int b) {
    int c, d;
    #pragma HLS BIND_OP variable=c op=mul impl=fabric latency=2
    c = a*b;
    d = a*c;
    return d;
}
```

fmul	fabric
fmul	meddsp
fmul	fulldsp
fmul	maxdsp
fmul	primitivedsp

## Kernel Optimization

- `pragma HLS aggregate`
- `pragma HLS alias`
- `pragma HLS disaggregate`
- `pragma HLS expression_balance`
- `pragma HLS latency`
- `pragma HLS performance`
- `pragma HLS protocol`
- `pragma HLS reset`
- `pragma HLS top`
- `pragma HLS stable`

## Function Inlining

- `pragma HLS inline`

## Interface Synthesis

- `pragma HLS interface`
- `pragma HLS stream`

## Task-level Pipeline

- `pragma HLS dataflow`
- `pragma HLS stream`

## Pipeline

- `pragma HLS pipeline`
- `pragma HLS occurrence`

## Loop Unrolling

- `pragma HLS unroll`
- `pragma HLS dependence`

## Loop Optimization

- `pragma HLS loop_flatten`
- `pragma HLS loop_merge`
- `pragma HLS loop_tripcount`

## Array Optimization

- `pragma HLS array_partition`
- `pragma HLS array_reshape`

## Structure Packing

- `pragma HLS aggregate`
- `pragma HLS dataflow`

## Resource Utilization

- `pragma HLS allocation`
- `pragma HLS bind_op`
- `pragma HLS bind_storage`
- `pragma HLS function_instantiate`

# HLS data types

- μπορείς με κλασικά *char*, *int*, *float*, *int64\_t*, ...
  - αλλά προτιμάς με συγκεκριμένο αριθμό bit
    - *arbitrary precision*\*: *ap\_int*, *ap\_uint*, *ap\_fixed*, ...
    - προσαρμογή & βελτιστοποίηση (*area*,  $f_{clk}$ , power)
  - ορίζεις μήκος σε ακέραιο+κλασματικό μέρος
    - *fixed*<W,I> : W=total, I=integer bits (+sign)
    - *float*<W,E> : W=total, E=exponent, W-E-1=frac.
  - ορίζεις τρόπο στρογγυλοποίησης+overflow
  - προσοχή στις μεταβλητές (δεν είναι SW!)
    - κλασσικά *float* και *double* ==> τεράστιο HW

```
float foo_f      = 3.1459;    → wire(foo_t) -> Float-to-Double Converter unit ->
float var_f = sqrt(foo_f);   HW →           -> Double-Precision Square Root unit ->
                                         -> Double-to-Float Converter unit -> wire(var_f)
```

```
1 #include <ap_fixed.h>
2 ...
3 ap_fixed<18,6,AP_RND> t1 = 1.5;
4 // 0b00'0001.1000'0000'0000 (0x01800)
5 ap_fixed<18,6,AP_RND> t2 = -1.5;
6 // 0b11'1110.1000'0000'0000 (0x3e800)
7 ...
```

float	ap_float<32,8>	AP_RND	Round to infinity
double	ap_float<64,11>	AP_RND_ZERO	Round to zero
half	ap_float<16,5>	AP_SAT	saturation
bfloat16	ap_float<16,8>	AP_WRAP	wrap around

παραδείγματα

```
#include "ap_int.h"      *βιβλιοθ. για C++
#include <ap_float.h>
ap_int<9> var1;
ap_uint<1024> var2;
ap_fixed<19,7,AP_TRN,AP_SAT> var3;
ap_fixed<2,0> a = -0.5; // sign+1frac
ap_ufixed<1,0> x = 0.5; // 0 or 0.5
typedef ap_float<18,6> my_float;
typedef unsigned char dout 2;
```

# Παράδειγμα: MatMult

πίνακες A και B έρχονται στην είσοδο (in1 και in2) μέσω *memory mapped axi4* πρωτοκόλλου. Θα φτιαχτεί κύκλωμα με address generator που θα διαβάζει από εξωτερική RAM

σπάμε τον A κυκλικά σε 16 RAM banks, ώστε να μπορούμε να διαβάζουμε μια γραμμή παράλληλα, αλλά τον B κατά μπλοκ για να διαβάζουμε μια στήλη. Μεγ.παραλληλία=16.

τοποθετούμε τα στοιχεία εισόδου στις μνήμες (flattened/1D, στον μερικό χώρο dimXdim). Τα readA και readB εκτελούνται σειριακά μεταξύ τους (για ταυτόχρονα βλ. #pragma dataflow).

**LOOP\_TRIPCOUNT:** κυρίως για εκτίμηση/ανάλυση αποτελεσμάτων (latency,...), όχι κατασκευαστική οδηγία (πχ, σαν το unroll)

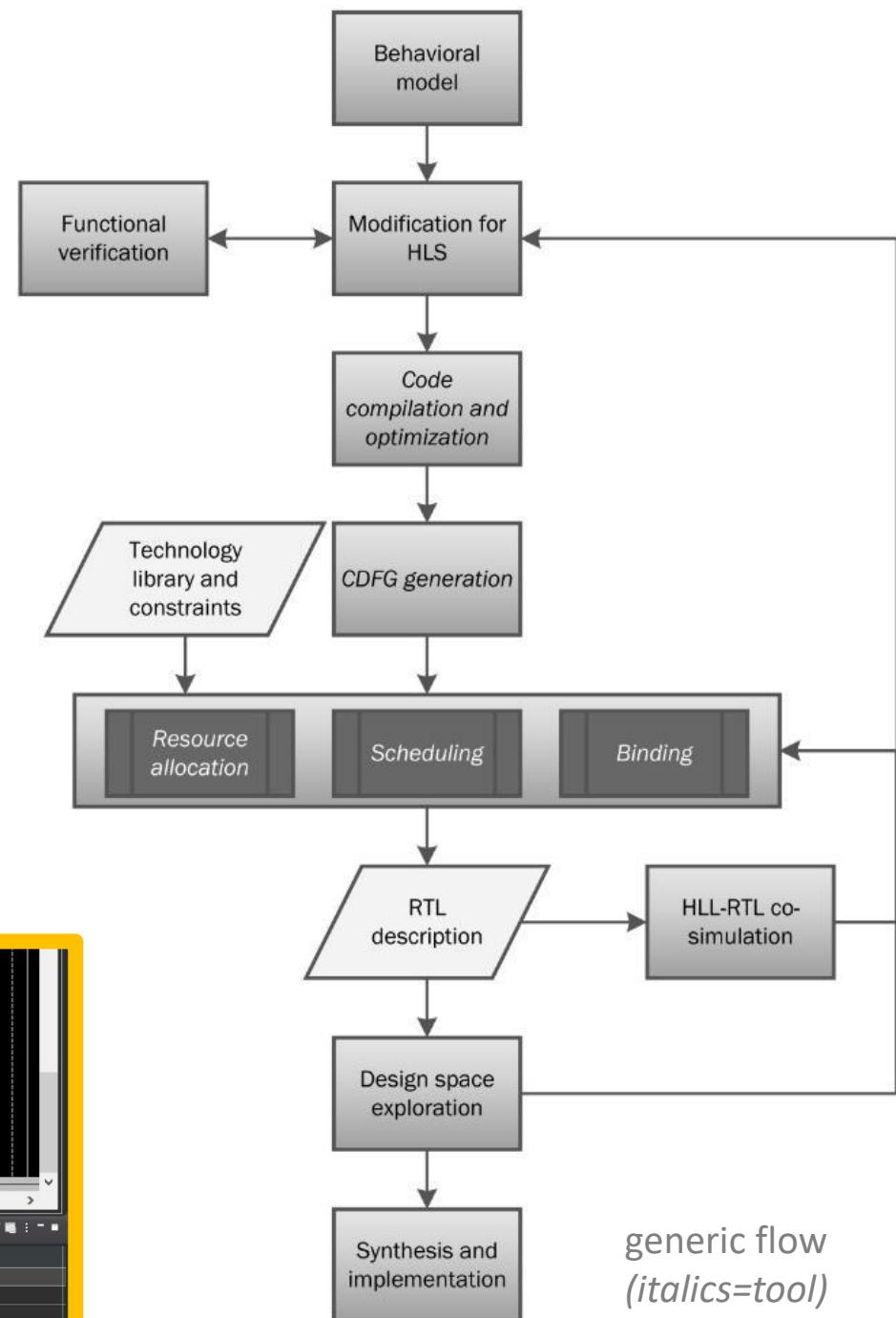
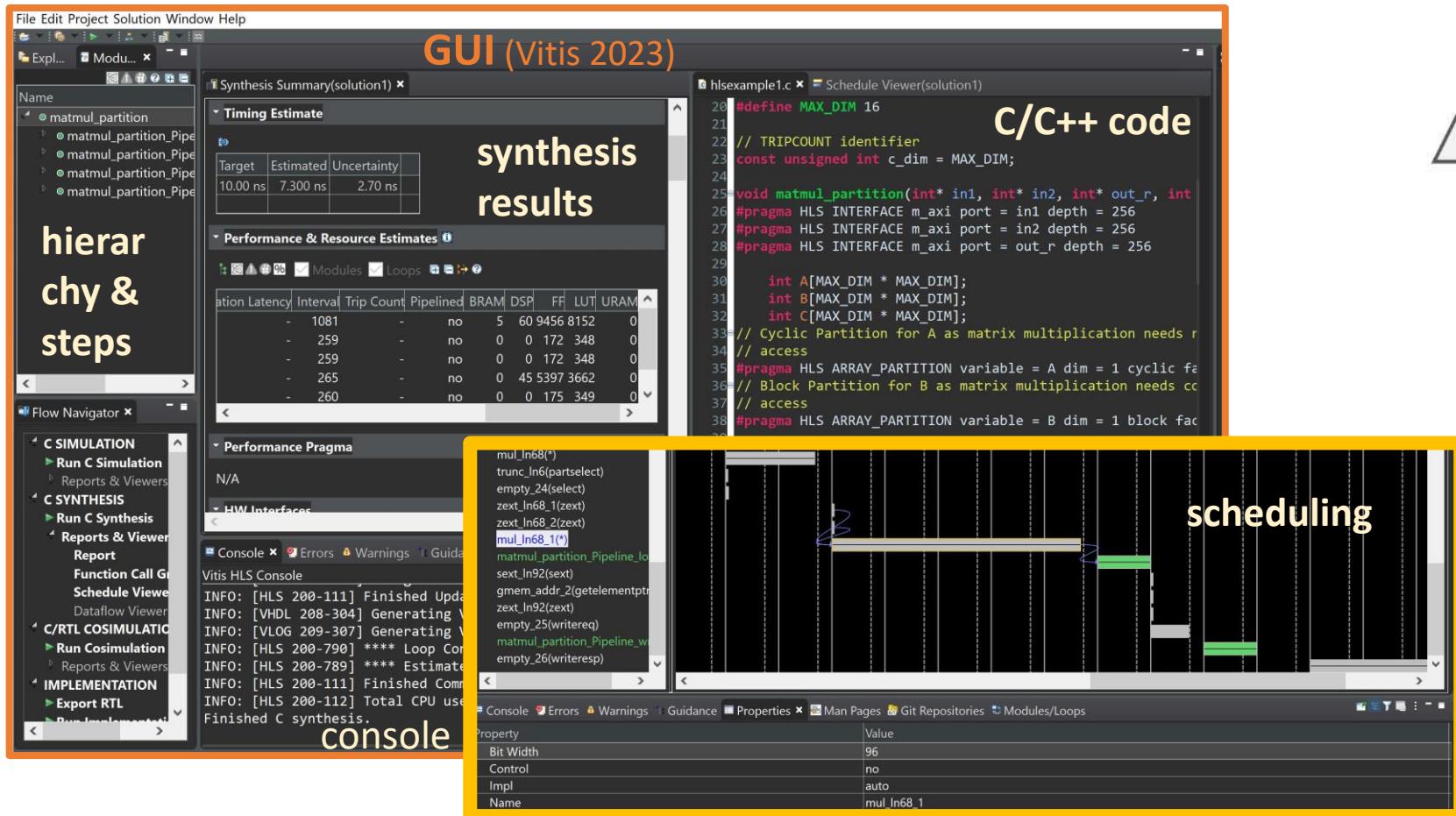
τριπλό loop εσωτερικού γινομένου ( $c_{ij} = a_{i1}*b_{1j} + a_{i2}*b_{2j} + \dots$ )

Τα δύο pragmas κατευθύνουν τον παραλληλισμό, πχ factor=8. Από κάποια έκδοση Vitis HLS, ακόμα κι αν αυτά παραλειφθούν παίρνουμε την μέγιστη/επιθυμητή παραλληλία αυτόματα (16 MAC), βλ. σχετικά VITIS "config\_unroll" σε script.tcl. Σε παλιότερα/άλλα HLS, ίσως χρειαστεί να γράψουμε τη συσσώρευση με πιο παράλληλο τρόπο (χωρίς εξάρτηση result μεταξύ iterations).

```
const unsigned int c_dim = MAX_DIM; // MAX_DIM=16 in ".h" !!!  
  
void matmul_partition(int* in1, int* in2, int* out_r, int dim, int rep_count) {  
#pragma HLS INTERFACE m_axi port = in1 depth = 256  
#pragma HLS INTERFACE m_axi port = in2 depth = 256  
#pragma HLS INTERFACE m_axi port = out_r depth = 256  
  
    int A[MAX_DIM * MAX_DIM];  
    int B[MAX_DIM * MAX_DIM];  
    int C[MAX_DIM * MAX_DIM];  
  
#pragma HLS ARRAY_PARTITION variable = A dim = 1 cyclic factor = 16  
#pragma HLS ARRAY_PARTITION variable = B dim = 1 block factor = 16  
  
readA: for (int itr = 0, i = 0, j = 0; itr < dim * dim; itr++, j++) {  
#pragma HLS LOOP_TRIPCOUNT min = c_dim*c_dim max = c_dim*c_dim  
    if (j == dim) { j = 0; i++; }  
    A[i * MAX_DIM + j] = in1[itr];  
}  
  
readB: for (int itr = 0, i = 0, j = 0; itr < dim * dim; itr++, j++) {  
#pragma HLS LOOP_TRIPCOUNT min = c_dim*c_dim max = c_dim*c_dim  
    if (j == dim) { j = 0; i++; }  
    B[i * MAX_DIM + j] = in2[itr];  
}  
  
loop2:  
    for (int x = 0; x < rep_count; x++) {  
#pragma HLS LOOP_TRIPCOUNT min = 1 max = 1  
        lreorder1:  
            for (int i = 0; i < dim; i++) {  
#pragma HLS LOOP_TRIPCOUNT min = c_dim max = c_dim  
            lreorder2:  
                for (int j = 0; j < dim; j++) {  
#pragma HLS LOOP_TRIPCOUNT min = c_dim max = c_dim  
                    int result = 0;  
                    lreorder3:  
                        for (int k = 0; k < MAX_DIM; k++) {  
#pragma HLS PIPELINE II=1  
#pragma HLS UNROLL factor=16  
                            result += A[i * MAX_DIM + k] * B[k * MAX_DIM + j];  
                        }  
                    }  
                    C[i * MAX_DIM + j] = result;  
    }  
}  
}  
  
writeC: for (int itr = 0, i = 0, j = 0; itr < dim * dim; itr++, j++) {  
#pragma HLS LOOP_TRIPCOUNT min = c_dim*c_dim max = c_dim*c_dim  
    if (j == dim) { j = 0; i++; }  
    out_r[itr] = C[i * MAX_DIM + j];  
}
```

# flow, tools, compiler,...

- **Vitis:** C++ file  $\rightarrow$  Clang/LLVM  $\rightarrow$  HLS S&B  $\rightarrow$  RTL  
(με βιβλιοθήκες, GUI, και "v++" linker)

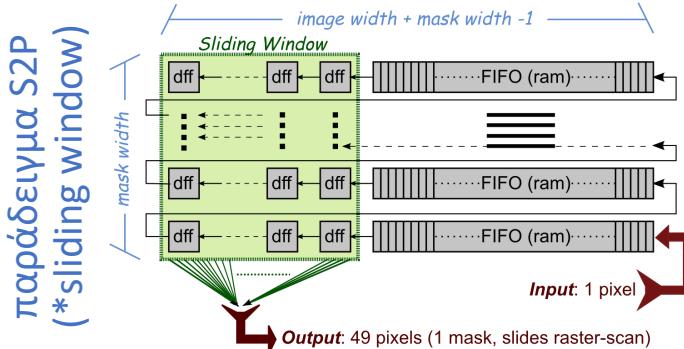


# Παράδειγμα: stencil

μεταβλητή `var_in` εμφανίζεται μόνιμα στην πόρτα και 2D πίνακας `in[][]` έρχεται με *memory mapped axi4* πρωτοκόλλο

εδώ έχουμε axilite πρωτόκολλο για να ρυθμίσει τις πόρτες (πχ, `offset` για `in[][]`), εκτός από το `m_axi` για το data burst. Επίσης, μοιράζονται δύο bus (ένα “`gmem`” κι ένα “`control`”) Το `var_in` έρχεται σαν `ctrlr` με axilite και μένει σε ένα register

**υπολογισμός 1D stencil** (σαν conv/ολίσθηση και τοπικές πράξεις για κάθε pixel, αλλά όχι απαραίτητα με συνεχές τετραγωνικό μοτίβο) (αντίστοιχα έχουμε και 2D stencil). Το pragma `stencil` βοηθά να φτιαχτεί sliding window\* buffer για serial-to-parallel μετατροπή στα data (πχ με shift registers ή/και FIFO για 2D stencil). Με αυτόν τον τρόπο παραλληλοποιείται ο υπολογισμός της μάσκας χωρίς περίπλοκο array partitioning. Κλασσική τεχνική.



με `pragma stencil`  
latency=10K cc  
DSP=8 , LUT=3K

χωρίς stencil  
latency=30K cc  
DSP=2 , LUT=2K

```
#include<string.h>
#define HS 100
#define WS 100
#define FW 3
#define FH 1

void top(float in[HS][WS], float out[HS][WS-FW+1], float var_in) {
    #pragma HLS INTERFACE m_axi port=in offset=slave bundle=gmem
    #pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem
    #pragma HLS INTERFACE s_axilite port=ctrlr bundle=control
    #pragma HLS INTERFACE s_axilite port=ctrlr bundle=control
    #pragma HLS INTERFACE s_axilite port=var_in bundle=control
    #pragma HLS INTERFACE s_axilite port=return bundle=control

    for (int i = 0; i < 100; i++) {
        for (int j = 0; j < 100-FW+1; j++) {
            #pragma HLS pipeline II=1
            #pragma HLS array_stencil variable=in
            out[i][j] = in[i][j] + in[i][j+1] + in[i][j+2] + var_in;
        }
    }
}
```

# Παράδειγμα: toyMLP

```

1 #include <ap_int.h>
2 #include <ap_fixed.h>
3 #include <hls_math.h>
4 #define N_IN    10 // inputs
5 #define N_HID   5  // hidden neurons
6 #define N_OUT   2  // output neurons
7
8 typedef ap_fixed<16, 8> data_t;
9 typedef ap_fixed<16, 8> weight_t;
10 typedef ap_fixed<16, 8> bias_t;
11
12 static data_t sigmoid(data_t x) {
13 #pragma HLS INLINE
14     return 1 / (1 + hls::exp(-x));
15 }
16
17 static void softmax(data_t in[2], data_t out[2]) {
18 #pragma HLS INLINE
19     data_t e0 = hls::exp(in[0]); data_t e1 = hls::exp(in[1]);
20     data_t sum = e0 + e1;
21     out[0] = e0 / sum; out[1] = e1 / sum;
22 }
23
24 static data_t neuron(const data_t in[N_IN], const weight_t w[N_IN],
25                      bias_t b, int n_inputs)
26 {
27 #pragma HLS INLINE off
28 #pragma HLS ARRAY_PARTITION variable=in complete dim=1
29 #pragma HLS ARRAY_PARTITION variable=w complete dim=1
30
31     data_t acc = b;
32
33 DOT:for (int i = 0; i < N_IN; i++) {
34 #pragma HLS UNROLL
35     if (i < n_inputs) acc += w[i] * in[i];
36 }
37
38 return sigmoid(acc);
39 }
```

κύρια συνάρτηση, με AXI stream για την είσοδο, lite για την έξοδο

σειριακή αποθήκευση εισόδου σε registers, για παράλληλο διάβασμα

fixed-point αριθ., για οικονομία (Q8.8)

*'inline'*: αντιγραφή κώδικα αντί κλήση συνάρτησης/comm.

1 neuron (6-11 βάρη + data)

hidden layer (5 neurons)

output layer (2 neurons)

```

38 void toyMLP(ap_int<8> in_stream[N_IN], data_t out_softmax[2]) {
39 #pragma HLS INTERFACE axis port=in_stream
40 #pragma HLS INTERFACE s_axilite port=out_softmax bundle=ctrl
41 #pragma HLS INTERFACE s_axilite port=return bundle=ctrl
42
43 #pragma HLS DATAFLOW
44
45     data_t inputs[N_IN];
46 #pragma HLS ARRAY_PARTITION variable=inputs complete dim=1
47 READ_PIX: for (int i = 0; i < N_IN; i++) {
48 #pragma HLS PIPELINE II=1
49         inputs[i] = (data_t) in_stream[i]; }
50
51 const weight_t W1[N_HID][N_IN] = { // WEIGHTS ROM (HIDDEN LAYER)
52     { 0.15, -0.20, 0.30, -0.25, 0.40, -0.10, 0.1, 0.2, 0.3, 0.4 },
53     { -0.35, 0.50, -0.45, 0.20, 0.10, 0.60, 0.5, 0.6, 0.7, 0.8 },
54     { 0.25, 0.15, 0.05, -0.10, 0.20, -0.30, 0.1, 0.2, 0.3, 0.0 },
55     { 0.25, 0.15, 0.05, -0.10, 0.20, -0.30, 0.5, 0.6, 0.7, 0.9 },
56     { 0.40, -0.40, 0.35, -0.35, 0.30, -0.30, 0.1, 0.2, 0.3, 0.5 } };
57 const bias_t B1[N_HID] = { 0.05, -0.02, 0.01, 0.3, 0.5 };
58 #pragma HLS ARRAY_PARTITION variable=W1 complete dim=0
59 #pragma HLS ARRAY_PARTITION variable=B1 complete dim=1
60
61     data_t hid[N_HID];
62 #pragma HLS ARRAY_PARTITION variable=hid complete dim=1
63 HIDDEN: for (int h = 0; h < N_HID; h++) {
64         hid[h] = neuron(inputs, W1[h], B1[h], N_IN); }
65
66 const weight_t W2[N_OUT][N_HID] = { // WEIGHTS ROM (OUTPUT LAYER)
67     { 0.55, -0.45, 0.35, 0.25, 0.01 },
68     { -0.30, 0.60, -0.20, 0.10, 0.02 } };
69 const bias_t B2[N_OUT] = { 0.0, 0.05 };
70 #pragma HLS ARRAY_PARTITION variable=W2 complete dim=0
71 #pragma HLS ARRAY_PARTITION variable=B2 complete dim=1
72
73     data_t logits[2];
74 #pragma HLS ARRAY_PARTITION variable=logits complete dim=1
75 OUTPUT: for (int o = 0; o < N_OUT; o++) {
76         logits[o] = neuron(hid, W2[o], B2[o], N_HID); }
77
78 softmax(logits, out_softmax);
79 }
```

# Παράδειγμα: toyMLP, αποτελέσματα+αλλαγές

- φτιάχνει 2 νευρώνες με pipelining  $ll=1$ 
  - ένα κρυφό, κι ένα εξόδου με κακό HW utilz.
  - παράλληλος μέσα, επανάχρηση στο layer
- το μεγάλο κόστος: *sigmoid* (και *softmax*)



## *sigmoid*

- extra neuron latency = 20 cc
- DSP=5, LUT=1K (x2 neurons)

αλλαγή σε *relu*, τότε *toyMLP*:

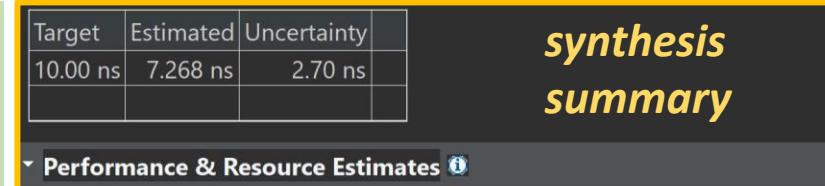
- latnc.=51, DSP=26, LUT=5K

## *softmax* (μόνο του)

- latency=5, DSP=6, LUT=2K

με **10-bit** arithmetic (vs 16b)

- *toyMLP* latency= 68 (vs 91)
- και DSP=22, FF=7K, LUT=5K
- για 5-bit: μικρή μείωση LUT



Performance & Resource Estimates	
Modules & Loops	Issue Violation Distance Slack Latency(cycles) Latency(ns) Iteration Latency Interval Trip Count Pipelined BRAM DSP FF LUT URAM
toyMLP	91 910.000 - 40 - dataflow 0 35 9107 6745 0
Loop_READ_PIX_proc14	- 12 120.000 - 12 - no 0 0 166 154 0
Loop_HIDDEN_proc	- 39 390.000 - 39 - no 0 13 2415 1561 0
HIDDEN	- 37 370.000 34 1 5 yes - - - - -
neuron	- 32 320.000 - 1 - yes 0 13 1962 1076 0
exp_17_9_s	- 5 50.000 - 1 - yes 0 3 527 448 0
Loop_OUTPUT_proc	- 31 310.000 - 31 - no 0 16 2665 1689 0
exp_16_8_1	- 6 60.000 - 6 - no 0 3 486 475 0
exp_16_8_s	- 6 60.000 - 6 - no 0 3 486 475 0
Block_for_end106112_proc	- 0 0.0 - 0 - no 0 0 32 29 0
Block_for_end106113_proc	- 29 290.000 - 29 - no 0 0 643 560 0

Name	DSP	Pragma	Variable	Op	Impl	Latency
toyMLP	35	-				
Loop_READ_PIX_proc14	-	-				
Loop_HIDDEN_proc	13					
HIDDEN						
add_ln80_fu_293_p2	-					
neuron	13					
mac_muladd_16s_8s_17s_23_4_1_U12	1					
mac_muladd_16s_8s_17s_23_4_1_U12	1					
mac_muladd_16s_9s_23s_24_4_1_U13	1					
mac_muladd_16s_9s_23s_24_4_1_U13	1					
mac_muladd_16s_8s_24ns_24_4_1_U14	1					
mac_muladd_16s_8s_24ns_24_4_1_U14	1					

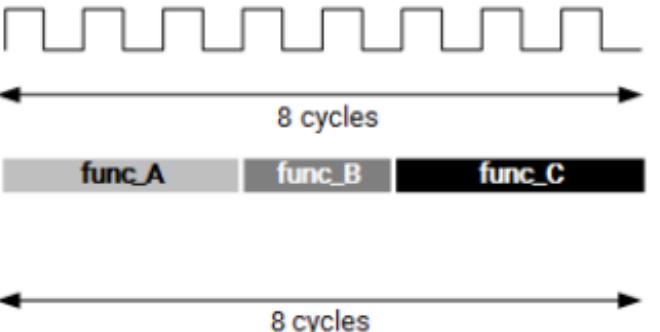
# Παρακάτω...

- πρέπει σκεφτούμε & δείξουμε παραλληλισμό (πχ σε *loop-carried dependencies*)
  - είτε με pragmas: *partition*, *dependence*, κ.α.
  - είτε με αλλαγή κώδικα (λιγότερο σήμερα...)
- σημαντικό το **data alignment** (μνήμη, I/O)
- task-level pipelining: **#pragma dataflow**
  - func* ξεκινά με πρώτα διαθέσιμα δεδομένα
  - πχ, *func* συνδέονται μεταξύ τους με FIFO\*

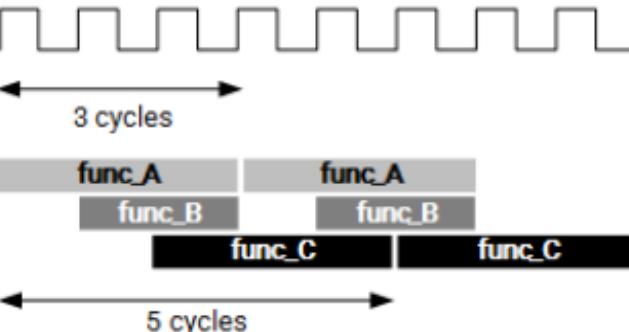
```
void top (a,b,c,d) {
    ...
    func_A(a,b,i1);
    func_B(c,i1,j2);
    func_C(i2,d)
    ...
    return d;
}
```

func_A	func_B	func_C
--------	--------	--------

χωρίς dataflow



με dataflow



```
struct One
{
    short int s;
    int i;
    char c;
}

struct Two
{
    int i;
    char c;
    short int s;
}
```



## Data Alignment:

πχ, GCC τοποθετεί τα data με τη σειρά και με padding

**Data Alignment:** για έλεγχο της στοίχισης/συμπύκνωσης (ταχύτητα vs χώρο), σε GCC βλ. `_attribute_((packed))`, σε HLS βλ. `pragma aggregate` και `pragma disaggregate`

# Συγκρίσεις HLS-HDL (2025)

- βιβλιογραφική μελέτη δημοσιεύσεων

S. Lahti & T. D. Hämäläinen. "High-level Synthesis for FPGAs...", 10.1109/ACCESS.2025.3540320

- περίπου 1:1:1 σε ταχύτητα (καλό:ισοπαλία:κακό)
- HDL μικρότερο κύκλωμα σε  $\frac{2}{3}$  περιπτώσεις
- HLS μείωση κόπου κατά 50-75% (LoC, time)
- vs 2019 HLS: -5% HW, +10%  $f_{clk}$ , -20% κόπος

- προσωπικές παρατηρήσεις/εμπειρία

- κόπος "καθοδήγησης" HLS προς βέλτιστο κύκλωμα ίσως γίνει εφάμιλλος του κόπου HDL
  - ειδικά για έμπειρους σε HDL, με προίκα αρχεία
- δίκαιη σύγκριση = δίκαιη/ίση προσπάθεια...
- SW complexity (+time/budget)  $\rightarrow$  προς HLS

TABLE 6. HLS vs. mRTL performance comparisons.

Metric	N	HLS/mRTL mean	Geom. std. dev.	HLS better or equal to mRTL
Max. clock freq.	70	0.97	1.65	53%
Latency	28	2.06	3.73	14%
Execution time	7	3.36	4.04	14%
Data rate	36	0.72	3.67	56%
Performance	79	0.67	3.04	41%

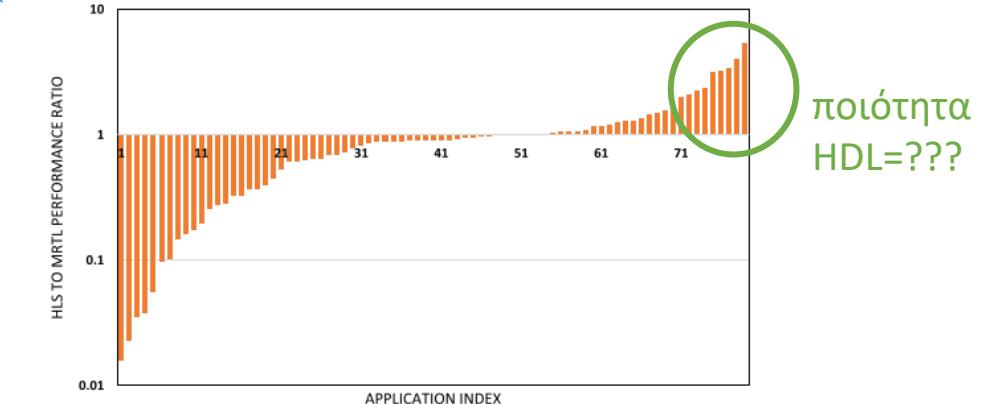
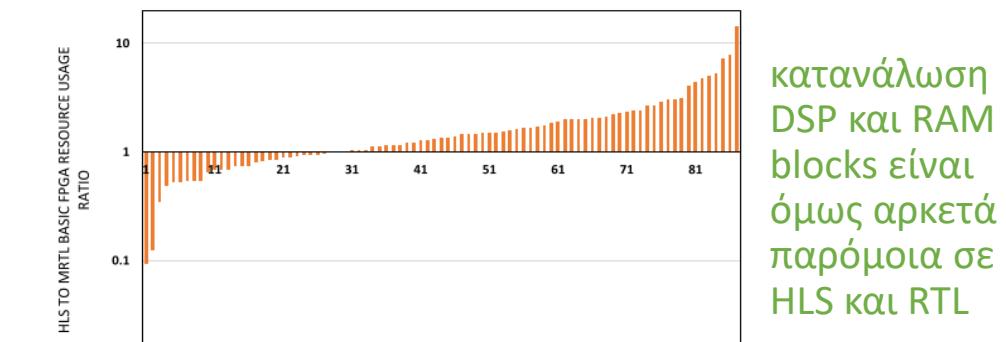


TABLE 7. HLS vs. mRTL Basic FPGA resource usage comparison.

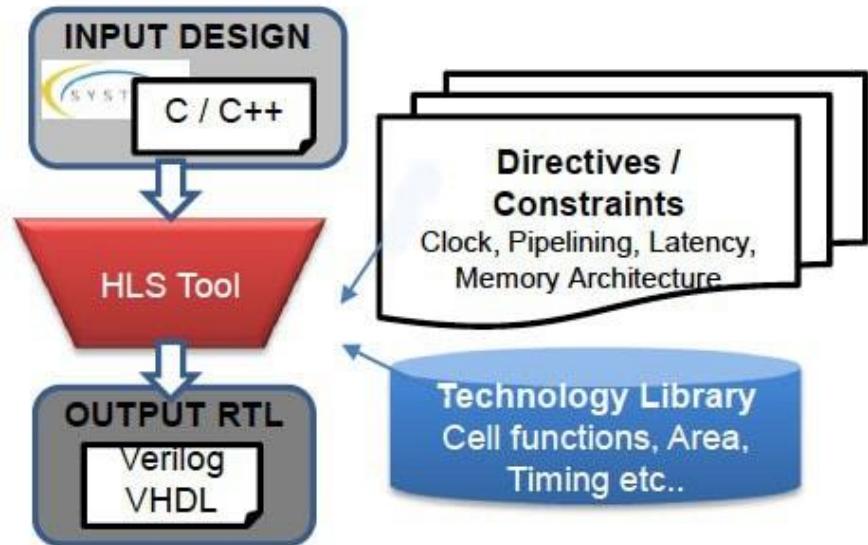
Metric	N	HLS/mRTL mean	Geom. std. dev.	HLS better or equal to mRTL
LUTs	79	1.27	2.29	37%
DFFs	73	1.28	2.28	37%
Slices	19	1.63	1.39	5%
Basic FPGA res.	87	1.36	2.15	33%



κατανάλωση DSP και RAM blocks είναι όμως αρκετά παρόμοια σε HLS και RTL

# Επίλογος (2025)

- Xilinx HLS και C/C++ επικρατούν στο θέμα
  - περίπου σε 70% εφαρμογών/δημοσιεύσεων
  - δεύτερο το MATLAB με <10% (βλ. *HDL coder*)
  - 12 εργαλεία σήμερα, μειώνονται με τα χρόνια
- HLS μέθοδος βελτιώνεται κι επεκτείνεται
  - καλή για γρήγορη ανάπτυξη κι εξερεύνηση
    - HDL για οικονομία υλικού (και ταχύτητα επεξργ?)
  - με μεγάλη διασπορά στα αποτελέσματα της
    - τυχαιότητα σε εργαλείο, μηχανικούς, εφαρμογές
- **Θέλει προσαρμογή κώδικα & γνώση HW**
  - χωρίς: δυναμική μνήμη, *pointers*, αναδρομή
  - με: *bit-widths*, *HW-friendly alg.*, παραλληλίες
  - ...



Benefits	Challenges
Reduced effort in design entry Improved verification process Enables DSE Easier reuse of designs Leveraging of software techniques Familiar input languages	Possibly lower QoR Complications in verification Difficulty of incremental changes Lack of standardization Limitations of the input languages Time-consuming DSE Lack of trained engineers