



ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΕΠΕΞΕΡΓΑΣΤΩΝ & ΥΛΙΚΟΥ
ΗΡΥ 302: ΟΡΓΑΝΩΣΗ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΑΡΙΝΟ ΕΞΑΜΗΝΟ 2020-2021

Εργασία #1: Χωρίζεται σε 3 φάσεις

1^η φάση: προτεινόμενος χρόνος ολοκλήρωσης 8 ημέρες

«Σχεδίαση μονάδας αριθμητικών και λογικών πράξεων (ALU), και αρχείου καταχωρητών (Register File)»

- Μελετήστε πρώτα καλά την εκφώνηση -

Σκοπός της 1^{ης} φάσης

Σχεδίαση σε VHDL μιας μονάδας αριθμητικών και λογικών πράξεων και ενός αρχείου καταχωρητών, και η προσομοίωση τους με το εργαλείο της Xilinx, ISE ή Vivado.

Προαπαιτούμενα

Καλή κατανόηση της VHDL, συμπεριφορική (behavioral) και δομική (structural), καθώς και του περιβάλλοντος της Xilinx. Γνώσεις που αποκτήθηκαν στην Προχωρημένη Λογική Σχεδίαση, στη Λογική Σχεδίαση, και στους Ψηφιακούς Υπολογιστές.

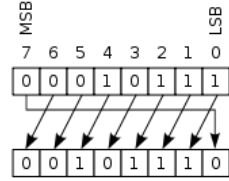
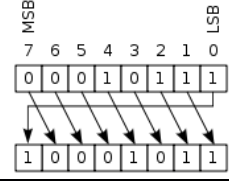
Διεξαγωγή

A) Μονάδα αριθμητικών και λογικών πράξεων (Arithmetic Logic Unit ή ALU)

Είναι συνδυαστικό κύκλωμα, δεν έχει ρολόι. Έχει τις εξής εισόδους και εξόδους:

Σήμα	Είδος -Πλάτος	Λειτουργία
A	είσοδος (32 bits)	Πρώτος τελεστής σε συμπλήρωμα ως προς 2
B	είσοδος (32 bits)	Δεύτερος τελεστής σε συμπλήρωμα ως προς 2
Op	είσοδος (4 bits)	Κωδικός πράξης
Out	έξοδος (32 bits)	Αποτέλεσμα σε συμπλήρωμα ως προς 2
Zero	έξοδος (1 bit)	Ενεργοποιημένη αν το αποτέλεσμα είναι μηδέν
Cout	έξοδος (1 bit)	Ενεργοποιημένη αν υπήρξε κρατούμενο εξόδου (Carry Out)
Onf	έξοδος (1 bit)	Ενεργοποιημένη αν υπήρξε υπερχείλιση

Η συμπεριφορά της ALU είναι η εξής:

Κωδικός	Πράξη	Αποτέλεσμα
Op = 0000	Πρόσθεση	Out = A + B
Op = 0001	Αφαίρεση	Out = A - B
Op = 0010	Λογικό “AND”	Out = A & B
Op = 0011	Λογικό “OR”	Out = A B
Op = 0100	Αντιστροφή του A	Out = ! A
Op = 0110	Λογικό “NAND”	Out = A !& B
Op = 1000	Αριθμητική ολίσθηση δεξιά κατά μια θέση. MSB ← [παλιό MSB]	Out= (int) A >> 1 Αποτέλεσμα = {A[31], A[31], ... A[1]}
Op = 1001	Λογική ολίσθηση δεξιά κατά μια θέση. MSB ← ‘0’	Out= (unsigned int) A >> 1 Αποτέλεσμα = {0, A[31], ... A[1]}
Op = 1010	Λογική ολίσθηση αριστερά κατά μια θέση. LSB ← ‘0’	Out= A << 1 Αποτέλεσμα = {A[30], A[29],... A[0],0}
Op = 1100	Κυκλική ολίσθηση (rotate) αριστερά το A κατά μια θέση	
Op = 1101	Κυκλική ολίσθηση (rotate) δεξιά το A κατά μια θέση	

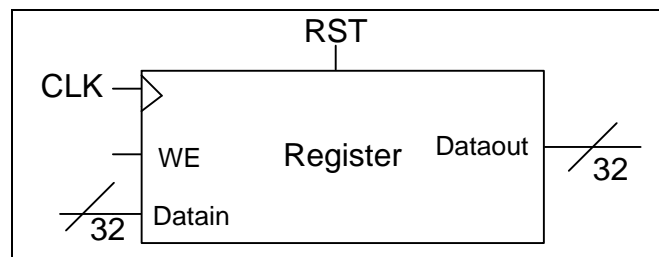
- Υλοποιήστε την ALU σε VHDL. Χρησιμοποιήστε Behavioral κώδικα όσο περισσότερο μπορείτε. Για παράδειγμα, για να φτιάξετε έναν αθροιστή χρησιμοποιήστε τον τελεστή «+» και για την αφαίρεση «-». Μην σχεδιάσετε κυκλώματα αθροιστών ξεκινώντας με τις βασικές λογικές πύλες, π.χ. μην φτιάξετε half-adder με XOR, AND, στη συνέχεια full-adder κλπ. Χρησιμοποιήστε τύπους δεδομένων (ακεραίους) που υπάρχουν σε «Standard VHDL Packages» π.χ. [ieee.std_logic_1164.all](#); [ieee.numeric_std.all](#); [ieee.std_logic_unsigned.all](#); [ieee.std_logic_signed.all](#); [ieee.std_logic_arith.all](#); κλπ. [Google it!](#)
- Προσθέστε καθυστέρηση στην τελική έξοδο της ALU χρησιμοποιώντας το **after** της VHDL ώστε το αποτέλεσμα να παράγεται 10 nanoseconds μετά την είσοδο, π.χ. outALU <= outAND **after** 10ns; [Google it.](#)
- Προσοχή στα σήματα εξόδου **Zero**, **Count** και **Ovf**. Ποιες είναι οι συνθήκες που ορίζουν τη συμπεριφορά αυτών των σημάτων; Τι εισόδους θα πρέπει να δώσετε στην προσομοίωση για να βεβαιωθείτε ότι τα σήματα αυτά λειτουργούν σωστά;
- Προσομοιώστε την ALU στο περιβάλλον της Xilinx. Είναι απαραίτητο να ελέγξετε όλες τις πράξεις της ALU με αρκετές διαφορετικές τιμές στις εισόδους ώστε να επαληθεύσετε την ορθή λειτουργία της σε όλες τις περιπτώσεις.

B) Αρχείο καταχωρητών (Register File ή RF)

Είναι κύκλωμα που δημιουργείται με καταχωρητές και συνδυαστικά κυκλώματα.

B1. Δημιουργία καταχωρητή

Υλοποιήστε αρχικά σε VHDL έναν καταχωρητή 32 bits. Θα έχει τα εξής σήματα: **ρολόι (CLK - 1 bit)**, **reset (RST - 1 bit)**, **δεδομένα εισόδου (Datain - 32 bits)**, **ένα σήμα για Write Enable (WE - 1 bit)**, και **δεδομένα εξόδου (Dataout - 32 bits)**. Η διεπαφή του register φαίνεται στην Εικόνα 1. Η έξοδος Dataout θα πρέπει να αλλάζει 10 nsec μετά τον θετικό παλμό ρολογιού. Αυτό υλοποιείται πάλι με χρήση του **after** που περιγράφηκε παραπάνω.



Εικόνα 1: Καταχωρητής πλάτους 32 bits

B2. Δημιουργία αρχείου καταχωρητών (RF)

Δημιουργήστε 32 καταχωρητές, όπως αυτόν που υλοποιήσατε στο βήμα B1. Προτείνεται να το κάνετε χρησιμοποιώντας το **for-generate** της VHDL. Στη συνέχεια, κάνοντας τη συνδεσμολογία που παρουσιάζεται στην Εικόνα 2 υλοποιήστε το RF με 32 καταχωρητές με τρεις θύρες, δύο ανάγνωσης και μία εγγραφής. Η διεπαφή του RF έχει τις εξής εισόδους και εξόδους:

Σήμα	Είδος -Πλάτος	Λειτουργία
Ard1	Είσοδος (5 bits)	Διεύθυνση πρώτου καταχωρητή για ανάγνωση
Ard2	Είσοδος (5 bits)	Διεύθυνση δεύτερου καταχωρητή για ανάγνωση
Awr	Είσοδος (5 bits)	Διεύθυνση καταχωρητή για εγγραφή
Dout1	Έξοδος (32 bits)	Δεδομένα πρώτου καταχωρητή
Dout2	Έξοδος (32 bits)	Δεδομένα δεύτερου καταχωρητή
Din	Είσοδος (32 bits)	Δεδομένα για εγγραφή
WrEn	Είσοδος (1 bit)	Ενεργοποίηση εγγραφής καταχωρητή
Clk	Είσοδος (1 bit)	Ρολόι

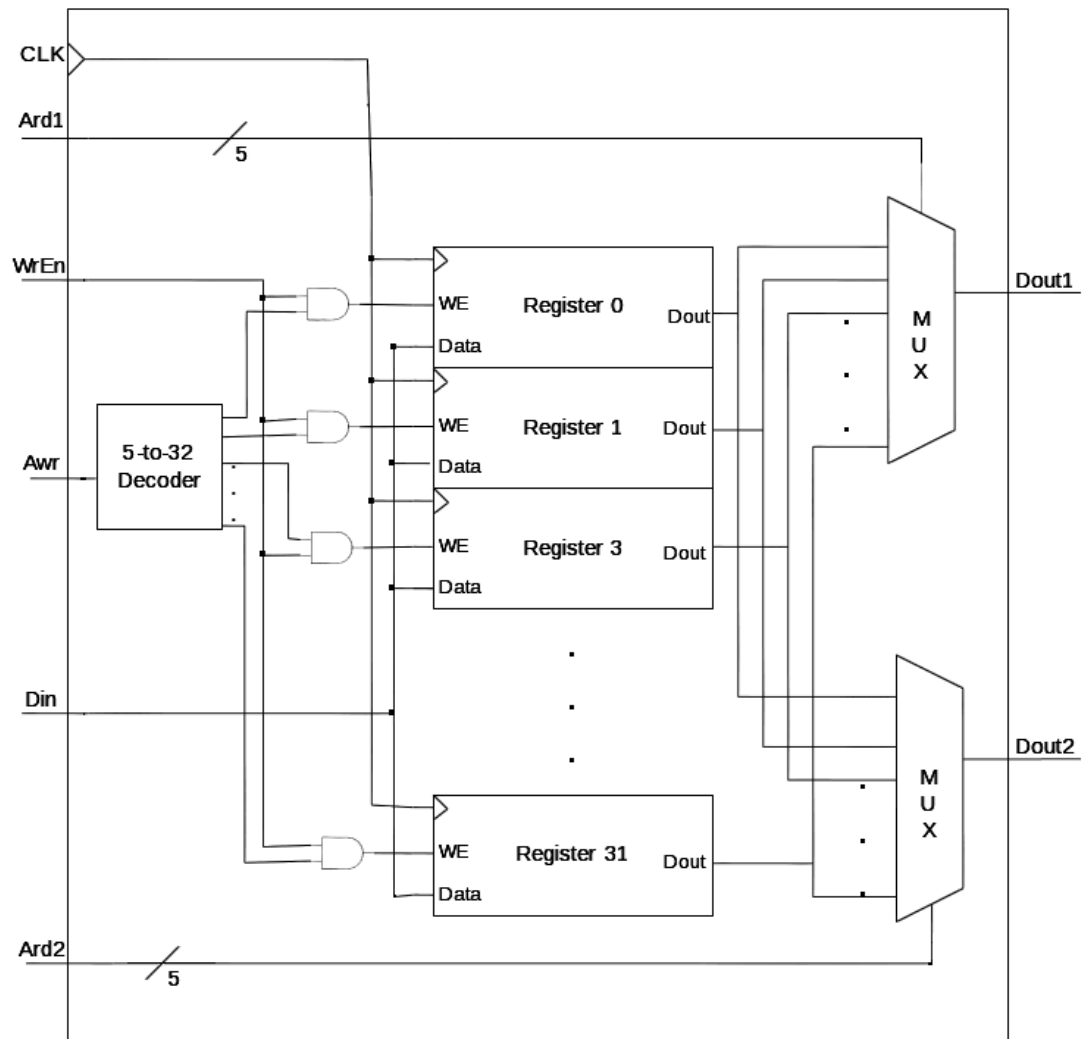
1. Υλοποιήστε σε VHDL το αρχείο καταχωρητών χρησιμοποιώντας τον καταχωρητή που υλοποιήσατε στο B1.
2. Στην διεπαφή δεν υπάρχει είσοδος ενεργοποίησης ανάγνωσης, το οποίο σημαίνει ότι το αρχείο καταχωρητών διαβάζει πάντα και από τις δύο θέσεις που υποδεικνύουν οι διευθύνσεις ανάγνωσης.
3. Τα κυκλώματα που απεικονίζονται στην Εικόνα 2 είναι συνδυαστικά, εκτός από τους registers.
4. Οι πολυπλέκτες και ο αποκωδικοποιητής να έχουν στην έξοδο τους καθυστέρηση 10 nsec

σε σχέση με τις εισόδους τους. Οι πύλες AND να έχουν καθυστέρηση 2 nsec σε σχέση με τις εισόδους τους. Για την καθυστέρηση χρησιμοποιείτε το **after** της VHDL που αναφέρθηκε παραπάνω.

5. Για τον καταχωρητή R0 θυμηθείτε ότι στον MIPS η τιμή του είναι πάντα σταθερή = '0'. Τι χρειάζεται να αλλάξετε στη συνδεσμολογία της Εικόνας 2 ώστε η τιμή του R0 να μην γράφεται ποτέ;
6. Προσομοιώστε την RF στο περιβάλλον της Xilinx. Δώστε πολλές διαφορετικές τιμές στις εισόδους ώστε να επαληθεύσετε την ορθή λειτουργία της σε όλες τις περιπτώσεις.
7. Στο testbench, θέστε την περίοδο του ρολογιού (CLK) **τουλάχιστον ίση με 100ns**, όχι λιγότερο.

Παραδοτέα (αφορούν όλη την Εργασία #1, όχι την 1^η φάση μόνο)

1. Προεργασία με block diagrams και λιτή τεχνική ανάλυση.
2. VHDL
3. Κυματομορφές προσομοίωσης. Θα σας βοηθήσει η χρήση του Waveform Configuration File (.wcfg) που προσφέρει η Xilinx, για ν' αποθηκεύετε τα σήματα σε αρχείο. Το αρχείο αυτό θα το κάνετε Load μετά από κάθε compile, για να εμφανίζονται αυτόματα τα σήματα που θέλετε να παρακολουθείτε.



Εικόνα 2: Αρχείο καταχωρητών (RF)

Γενικές οδηγίες – αφορούν όλες τις φάσεις της εργασίας

- Αν πρόκειται να χρησιμοποιήσετε δικό σας υπολογιστή μόνο, ακόμη και κατά την προσέλευσή σας στο εργαστήριο π.χ. στο laptop σας, μπορείτε να δουλέψετε αποκλειστικά με το Xilinx Vivado. Στους υπολογιστές του εργαστηρίου και του μηχανογραφικού όμως διατίθεται η Xilinx ISE 13.4, και αν πρόκειται να τους χρησιμοποιήσετε πρέπει να εξοικειωθείτε με τη συγκεκριμένη έκδοση. Σημειώνεται πως όλες οι εκδόσεις του Xilinx ISE είναι διαθέσιμες στην επίσημη σελίδα της εταιρείας.
- Στο courses υποβάλλετε μαζί με τους κώδικες VHDL, και το αρχείο με το configuration της κυματομορφής προσομοίωσης. Λέγεται Waveform Configuration file (.wcfg). Ψάξτε πως δημιουργείται, είναι πολύ εύκολο και θα σας «λύσει τα χέρια» στη δουλειά σας. Σας επιτρέπει τη δημιουργία αρχείου στο οποίο αποθηκεύονται τα σήματα που θέλετε να παρακολουθήσετε στην προσομοίωση. Οπότε κάθε φορά που κάνετε compile, έπειτα απλά κάνετε Load το αρχείο αυτό, και εμφανίζονται τα σήματα που θα παρακολουθήσετε. Το αρχείο .wcfg που θα δημιουργήσετε θα πρέπει να περιλαμβάνει τα σήματα που πρόκειται να μας δείξετε στις κυματομορφές σας. Θα αποφύγουμε έτσι τις καθυστερήσεις μέσα στο εργαστήριο – δεν θ' αναλώνεται χρόνος ψάχνοντας σήματα για να τα «τραβήξετε» μέσα στην κυματομορφή προσομοίωσης. Δείτε επίσης μήπως σας βολεύει το Waveform Database file (.wdb). [Google it](#).
- Ο προσομοιωτής της Xilinx λέγεται **ISim**, είναι ο ίδιος και στο ISE και στο Vivado. Υπάρχει λοιπόν συμβατότητα μεταξύ ISE-Vivado όσον αφορά τον προσομοιωτή. Έτσι, αν για παράδειγμα δουλέψετε στο σπίτι σας σε Vivado, project/προσομοίωση/δημιουργία .wcfg, μπορείτε αυτό το υλικό να το χρησιμοποιήσετε για να δημιουργήσετε project στο ISE 13.4 των υπολογιστών του εργαστηρίου και του μηχανογραφικού. Προσοχή όμως, θα πρέπει να εξοικειωθείτε με το ISE 13.4. Δεν είναι δύσκολο - μελετήστε το εγχειρίδιο του Xilinx ISE καθώς και τις συχνές ερωτήσεις στο courses.
- Θα κάνετε "Behavioral simulation". Είναι προσομοίωση σε αρχικό στάδιο της σχεδιαστικής ροής. Συγκεκριμένα, επιλέξτε "Behavioral simulation", δείτε πως το κάνουμε αυτό στις σελ. 6-7 στο εγχειρίδιο του Xilinx ISE στο courses. Θα βρείτε μια drop-down list κάτω από το "Simulation", εκεί επιλέξτε "Behavioral". Με την επιλογή "Implementation" μην ασχοληθείτε καθόλου. Οπότε μην κάνετε "Implement Design", "Place & Route", ούτε καν "Synthesize". Με τις παραπάνω ενέργειες μειώνεται πολύ ο χρόνος επεξεργασίας του PC, και ο χρόνος προσομοίωσης. Ο κώδικας σας θέλουμε να είναι όσο πιο φορητός (portable) γίνεται, και να μην αναλώνεται χρόνος στο εργαστήριο κατά τη δημιουργία του project και για εισαγωγή σημάτων στις κυματομορφές προσομοίωσης.
- Το **after** της VHDL δεν είναι synthesizable. Συγκεκριμένα, αν περάσετε τον κώδικα σας από "Synthesis" ή "Implement" τα **after** θα αγνοηθούν. Ο λόγος που το χρησιμοποιούμε όμως είναι ιδιαίτερα σημαντικός: για να κατανοήσετε ότι αν υλοποιήσουμε πραγματικό κύκλωμα σε τσιπ, **η έξοδος καθυστερεί σε σχέση με την είσοδο**. Αν τώρα για κάποιο λόγο ένα κύκλωμα σας θελήσετε να το περάσετε από "Synthesis" ή ακόμη και "Implement Design", π.χ. για να ελέγξετε πως ο κώδικας που έχετε γράψει είναι synthesizable δηλαδή μπορεί να μπει σε τσιπ, τα **after** θα αγνοηθούν και θα προστεθούν

απλά οι πραγματικές καθυστερήσεις, π.χ. η πραγματική καθυστέρηση για να “περάσει” η είσοδος ενός πολυπλέκτη στην έξοδο τουφ.

- Γράψτε “καθαρό κώδικα” VHDL, με τα modules ιεραρχικά οργανωμένα, π.χ. topModule, πολυπλέκτες, κωδικοποιητές, συγκριτές, ALU, register, RF, testbench. Μόνο αυτή η προσέγγιση θα σας οδηγήσει στο να συνδέσετε σωστά μεταξύ τους τα modules, και τελικά να ολοκληρώσετε επιτυχημένα τη σχεδιάσή σας.
- Όταν γράφετε κώδικα με **if..then** διερευνήσετε αν είναι απαραίτητο να κλείσετε όλα τα **else**. Μπορεί να είναι απαραίτητο, μπορεί και όχι.
- Αν κάποια πράγματα στην εκφώνηση δεν σας φαίνονται πλήρη ή ξεκάθαρα, πάρτε πρωτοβουλία και κάνετε ότι χρειάζεται, αιτιολογώντας το όμως. Μπορεί να υπάρχουν ελλείψεις στις εκφωνήσεις – στείλτε μας μήνυμα σχετικά - π.χ. να μας έχει ξεφύγει να γράψουμε κάποιο σήμα σε μια διεπαφή όπως CLK ή RST. Θεωρούμε ότι τα γνωρίζετε αυτά, π.χ. ότι ένα flip-flop έχει CLK και RST. Ένας register έχει σίγουρα CLK, δεν είναι όμως απαραίτητο να έχει και RST στην διεπαφή του.
- Συνδυαστικά κυκλώματα είναι αυτά που δεν έχουν κατάσταση και κατά συνέπεια ούτε ρολόι π.χ. μια λογική πύλη, ή ένα σύνολο από πύλες όπως είναι η ALU, ο πολυπλέκτης, ο κωδικοποιητής, ο αθροιστής. Αυτά λέγονται και κυκλώματα συνδυαστικής λογικής. Σύγχρονα ακολουθιακά κυκλώματα είναι το flip-flop και ο καταχωρητής, τα οποία έχουν ρολόι. Ασύγχρονα ακολουθιακά κυκλώματα είναι αυτά που δεν έχουν ρολόι αλλά έχουν κατάσταση π.χ. το latch.
- Σήμα RST αρχείου καταχωρητών (RF): δεν είναι απαραίτητο και δεν υπάρχει στην διεπαφή της εκφώνησης. Μπορείτε όμως αν θέλετε να το προσθέσετε. Σχεδιάστε το να είναι σύγχρονο ως προς το ρολόι, και ενεργό στο **‘1’ (active high)**.
- Σχετικά με τις καθυστερήσεις στις εξόδους των υποκυκλωμάτων, δηλ. το **after** της VHDL: την βάζετε μια φορά, π.χ. μόνο στις τελικές εξόδους των modules. Για παράδειγμα, στην υλοποίηση της ALU, η καθυστέρηση των 10ns στην έξοδο δεν πρέπει να μπει στην έξοδο του κάθε υπό-module που κάνει την πράξη, π.χ. και στην αφαίρεση, και στην πρόσθεση, και στο AND κλπ, αλλά χρειάζεται να μπει μόνο στην έξοδο του πολυπλέκτη που επιλέγει τι θα βγει στην έξοδο (δηλαδή μόνο στα σήματα Output, Onf, Cout, Zero). Ακολουθείστε την ίδια πρακτική όταν βάζετε καθυστερήσεις σ’ ένα κύκλωμα κωδικοποιητή, συγκριτή, πολυπλέκτη, register κλπ. Δηλαδή βάζετε 10ns καθυστέρηση μόνο στην τελική έξοδο του module.
- Μετά την 1^η φάση της εργασίας#1, καθυστερήσεις (**after**) να προσθέτετε μόνο όταν είναι απαραίτητο, π.χ. αν φτιάξετε έναν νέο MUX που θα τον συνδέσετε σε σημείο στο οποίο δεν υπάρχουν σε σειρά καθόλου άλλα κυκλώματα που να χουν καθυστέρηση στην έξοδο. Ο λόγος: διότι αν κάποιο σήμα εξαρτάται από περισσότερο του ενός «κουτιά» σε σειρά, η σχεδίαση τότε θα «αντιληφθεί» περισσότερη καθυστέρηση. Και αν αυτό το κάνουμε σε κάθε έξοδο, π.χ. σε ένα μικρό υποκύκλωμα που αποτελείται από λίγες πύλες, θα φτάσουμε να χουμε critical path που θα ναι μεγαλύτερο από την περίοδο του ρολογιού, 100ns!

- Σημαντικό τμήμα της εξέτασης αποτελούν οι κυματομορφές προσομοίωσης, στις οποίες πρέπει να φαίνονται ευδιάκριτα τα κύρια σήματα που πρέπει να παρακολουθήσουμε. Προτείνεται χρήση διαφορετικών χρωμάτων στις κυματομορφές και αλλαγή radix όπου χρειάζεται. Συγκεκριμένα, καλώδια/τιμές με πολλά bits θα πρέπει να δείχνονται σε δεκαεξαδικό (HEX) ή δεκαδικό (DEC) ώστε να έχουν λιγότερα ψηφία και να διευκολύνουν στην παρακολούθηση. Ανάλογα ότι βολεύει.
- Η δημιουργία του testbench είναι εξίσου σημαντική με τα κυκλώματα που φτιάχνετε. Αν χρειαστεί προκαλέστε καθυστέρηση - χρησιμοποιώντας το **after** - στα σήματα που δημιουργείτε στο testbench, σε αυτά δηλαδή που είναι είσοδοι προς το topLevel σας.
- Στο testbench της RF για την εγγραφή: αν διαφορετικά σήματα εισόδου φτάσουν σε διαφορετικές χρονικές στιγμές δεν μας πειράζει αφού η εγγραφή θα γίνει στην ακμή του ρολογιού. Η ανάγνωση της RF γίνεται συνδυαστικά, οπότε όταν αλλάζει η διεύθυνση του καταχωρητή, μετά από λίγο αλλάζει και η έξοδος.
- Επαληθεύστε τη λειτουργία κάθε **υποκυκλώματος, βήμα-προς-βήμα καθώς τα σχεδιάζετε**, π.χ. φτιάξτε testbench για έναν πολυπλέκτη και προσομοιώστε τον, το ίδιο κάνετε για την ALU, το ίδιο για έναν συγκριτή, για έναν κωδικοποιητή, για ένα flip-flop, για έναν καταχωρητή κλπ. Ελέγξτε λοιπόν κάθε υποκύκλωμα ξεχωριστά πρώτα, με δικό του testbench. Μετά, ενώστε αυτά που πρέπει σε ένα topLevel, φτιάξτε το αντίστοιχο testbench και προσομοιώστε. Δεν χρειάζεται βέβαια να φτιάχνετε testbench για πολύ μικρά κυκλώματα, π.χ. για να επαληθεύσετε μια πύλη AND.
- Η σύνδεση των modules γίνεται ιεραρχικά, δηλ. έχουμε modules που συνδέονται μεταξύ τους μέσα σε άλλο module που βρίσκεται σε 1 παραπάνω επίπεδο. Αυτό με τη σειρά του μπορεί να συνδέεται με άλλα modules, μέσα σε module που βρίσκεται σε ακόμη παραπάνω επίπεδο κ.ο.κ. Ακόμα και αν δεν συνδέονται μεταξύ τους τα modules, χρειάζεται να φτιάχνετε 1 module σε 1 παραπάνω επίπεδο το οποίο περιέχει τα χαμηλότερα σε ιεραρχία modules, ή, τα instances τους (αυτό συμβαίνει στις περισσότερες περιπτώσεις). Instances φτιάχνουμε με **component/port map**.
- Δημιουργείτε topLevel (ή ονομάστε το topModule) στο οποίο θα συνδέετε όλα τα σήματα που θέλετε να «δείτε» στην προσομοίωση. Φτιάξτε testbench, και εκεί κάνετε instance το topLevel.

2^η φάση: χρόνος ολοκλήρωσης 12 ημέρες

«Σχεδίαση των βασικών βαθμίδων του datapath ενός απλού επεξεργαστή»

- Μελετήστε πρώτα καλά την εκφώνηση -

Σκοπός της 2^{ης} φάσης

1. Ορισμός της αρχιτεκτονικής συνόλου εντολών (Instruction Set Architecture ή ISA)
2. Σχεδίαση της βαθμίδας ανάκλησης εντολών (Instruction Fetch ή IF)
3. Σχεδίαση της βαθμίδας αποκωδικοποίησης εντολών (Instruction Decoding ή ID)
4. Σχεδίαση της βαθμίδας εκτέλεσης εντολών (Execution ή EX)
5. Σχεδίαση της βαθμίδας πρόσβασης μνήμης (Memory ή MEM)

Αρχιτεκτονική Συνόλου Εντολών

Θα υλοποιήσετε τμήματα ενός non-pipelined επεξεργαστή βασισμένου σε υποσύνολο της αρχιτεκτονικής συνόλου εντολών CHARIS (CHAnia Risc Instruction Set), που αποτελείται από τα εξής:

1. 32 καταχωρητές, πλάτους 32 bits ο καθένας. Ο καταχωρητής R0 είναι πάντα μηδέν.
2. 32 bits πλάτος εντολών, με μέγεθος και θέση πεδίων που φαίνονται παρακάτω.
3. Εντολές αριθμητικών και λογικών πράξεων, π.χ. add, sub, nand, not, or, sra, sll, srl, ror, rol, li, addi, nandi, ori.
4. Εντολές διακλάδωσης: b, beq, bne.
5. Εντολές μνήμης: lb, sb, lw, sw.

Οι εντολές έχουν δύο τύπους format:

6-bits	5-bits	5-bits	5-bits	5-bits	6-bits
Opcode	rs	rd	rt	not-used	func

6-bits	5-bits	5-bits	16-bits
Opcode	rs	rd	Immediate

Η διευθυνσιοδότηση της μνήμης γίνεται με διευθύνσεις byte, και οι εντολές και τα δεδομένα (των εντολών lb, sb, lw και sw) πρέπει να είναι ευθυγραμμισμένα σε πολλαπλάσια των 4 bytes. Οπότε πρέπει να παράγετε τις διευθύνσεις ως διευθύνσεις byte, και μετά να δώσετε σωστή διεύθυνση στην μνήμη. Εφόσον η μνήμη είναι οργανωμένη σε 4-αδες byte, θα πρέπει να της δώσετε διεύθυνση λέξης την οποία θα "παράγετε" από την διεύθυνση byte.

Η κωδικοποίηση των εντολών γίνεται σύμφωνα με τον ακόλουθο πίνακα:

Opcode	FUNC	ΕΝΤΟΛΗ	ΠΡΑΞΗ
100000	110000	add	$RF[rd] \leftarrow RF[rs] + RF[rt]$
100000	110001	sub	$RF[rd] \leftarrow RF[rs] - RF[rt]$
100000	110010	nand	$RF[rd] \leftarrow RF[rs] \text{ NAND } RF[rt]$
100000	110100	not	$RF[rd] \leftarrow \neg RF[rs]$
100000	110011	or	$RF[rd] \leftarrow RF[rs] \mid RF[rt]$
100000	111000	sra	$RF[rd] \leftarrow RF[rs] \gg 1$
100000	111001	sll	$RF[rd] \leftarrow RF[rs] \ll 1$ (Logical, zero fill LSB)
100000	111010	srl	$RF[rd] \leftarrow RF[rs] \gg 1$ (Logical, zero fill MSB)
100000	111100	rol	$RF[rd] \leftarrow \text{Rotate left}(RF[rs])$
100000	111101	ror	$RF[rd] \leftarrow \text{Rotate right}(RF[rs])$
111000	-	li	$RF[rd] \leftarrow \text{SignExtend}(Imm)$
111001	-	lui	$RF[rd] \leftarrow Imm \ll 16$ (zero-fill)
110000	-	addi	$RF[rd] \leftarrow RF[rs] + \text{SignExtend}(Imm)$
110010	-	nandi	$RF[rd] \leftarrow RF[rs] \text{ NAND } \text{ZeroFill}(Imm)$
110011	-	ori	$RF[rd] \leftarrow RF[rs] \mid \text{ZeroFill}(Imm)$
111111	-	b	$PC \leftarrow PC + 4 + (\text{SignExtend}(Imm) \ll 2)$
000000	-	beq	if ($RF[rs] == RF[rd]$) $PC \leftarrow PC + 4 + (\text{SignExtend}(Imm) \ll 2)$ else $PC \leftarrow PC + 4$
000001	-	bne	if ($RF[rs] != RF[rd]$) $PC \leftarrow PC + 4 + (\text{SignExtend}(Imm) \ll 2)$ else $PC \leftarrow PC + 4$
000011	-	lb	$RF[rd] \leftarrow \text{ZeroFill}(31 \text{ downto } 8) \& \text{MEM}[RF[rs] + \text{SignExtend}(Imm)](7 \text{ downto } 0)$
000111	-	sb	$\text{MEM}[RF[rs] + \text{SignExtend}(Imm)] \leftarrow \text{ZeroFill}(31 \text{ downto } 8) \& RF[rd](7 \text{ downto } 0)$
001111	-	lw	$RF[rd] \leftarrow \text{MEM}[RF[rs] + \text{SignExtend}(Imm)]$
011111	-	sw	$\text{MEM}[RF[rs] + \text{SignExtend}(Imm)] \leftarrow RF[rd]$

Διεξαγωγή

A. Μελετήστε την κωδικοποίηση των εντολών του CHARIS

Παρατηρήστε την ομαδοποίηση τους έχοντας στο μυαλό σας τον σκοπό της αποκωδικοποίησης : να παραχθούν τα απαραίτητα σήματα ελέγχου για την εκτέλεση της κάθε εντολής. Τέτοια σήματα είναι ο κωδικός πράξης της ALU, οι διευθύνσεις ανάγνωσης και τυχόν εγγραφής στο Register File, η επιλογή δύο καταχωρητών, ή, η επιλογή καταχωρητή και Immediate για μια πράξη κ.α.

Η επιλογή κωδικών των εντολών έγινε έτσι ώστε η αποκωδικοποίηση τους να είναι σχετικά απλή. Βρείτε τις υπάρχουσες συμμετρίες ώστε να απλοποιήσετε τη λογική αποκωδικοποίησης.

B. Υλοποίηση κύριας μνήμης 2048x32

Η κύρια μνήμη του CHARIS είναι όμοια με αυτή του MIPS. Περιέχει τις **εντολές και τα δεδομένα του εκάστοτε προγράμματος** που εκτελείται. Χωρίζεται σε δύο τμήματα (segments): το **text segment** στο οποίο αποθηκεύονται οι εντολές, και το **data segment** στο οποίο αποθηκεύονται τα δεδομένα. Το text segment ξεκινάει από τη διεύθυνση **0x000**, και το data segment ξεκινάει από τη διεύθυνση **0x400**. Για περισσότερες πληροφορίες σχετικά με την κατάτμηση της μνήμης ανατρέξτε σε ύλη μαθήματος προηγούμενου εξαμήνου.

Χρησιμοποιήστε τον κώδικα της Εικόνας 1 για να φτιάξετε μια μνήμη RAM 2048 θέσεων, των 32 bits η κάθε θέση. Έχει μια θύρα ανάγνωσης για τις εντολές, και μια θύρα ανάγνωσης/εγγραφής για τα δεδομένα. Η διεπαφή της έχει τα εξής σήματα:

Σήμα	Πλάτος	Είδος	Λειτουργία
clk	1bit	είσοδος	ρολόι
inst_addr	11 bits	είσοδος	διεύθυνση εντολής
inst_dout	32 bits	έξοδος	εντολή που διαβάστηκε από την μνήμη
data_we	1 bit	είσοδος	σημαία (flag) ενεργοποίησης εγγραφής στη μνήμη
data_addr	11 bits	είσοδος	διεύθυνση για ανάγνωση/εγγραφή δεδομένων
data_din	32 bits	είσοδος	δεδομένα που θα εγγραφούν στη μνήμη
data_dout	32 bits	είσοδος	δεδομένα που διαβάστηκαν από τη μνήμη

Σημείωση 1: Η μνήμη χρειάζεται αρχικοποίηση. Αυτό γίνεται με το κομμάτι κώδικα της Εικόνας 1 που επισημαίνεται με έντονα γράμματα. Ο συγκεκριμένος κώδικας “διαβάζει” το αρχείο **rom.data** και φορτώνει κάθε γραμμή του σε μια διαφορετική διεύθυνση. Προσοχή: το **rom.data** πρέπει να το βάλετε στον ίδιο φάκελο με το project σας.

Σημείωση 2: Σε αυτή τη φάση της εργασίας χρειάζεστε δύο instances του module της μνήμης, ένα για να το συνδέσετε με τη βαθμίδα IF, και ένα για να το συνδέσετε με τη βαθμίδα MEM. Για τη βαθμίδα IF τα σήματα **data_we**, **data_addr**, **data_din**, **data_dout** της μνήμης δεν χρησιμοποιούνται. Για τη βαθμίδα MEM δε χρησιμοποιούνται τα σήματα **inst_addr**, **inst_dout**. Τις χρησιμοποιήσιμες εισόδους πρέπει να τις συνδέσετε με **0** στο **port map**, ενώ τις χρησιμοποιήσιμες εξόδους μπορείτε να τις αφήσετε ασύνδετες ή ακόμη καλύτερα χρησιμοποιήστε το “=>open” της VHDL στο **port map**.

Σημείωση 3: Σε αυτή τη φάση της εργασίας λοιπόν θα συνδέσετε μια μνήμη με τη βαθμίδα IF, και άλλη μια μνήμη με τη βαθμίδα MEM. Όπως αναφέρθηκε παραπάνω αυτό γίνεται με 2

instances, με **component/port map**. Θα κάνετε ξεχωριστό testbench για την κάθε περίπτωση ώστε να επαληθεύσετε ότι κάνατε σωστά τις συνδέσεις (σε επόμενη φάση της εργασίας θα έχουμε 1 μόνο instance της μνήμης συνδεδεμένο και με τις δυο βαθμίδες)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use std.textio.all;
use ieee.std_logic_textio.all;

entity RAM is
    port (
        clk          : in std_logic;
        inst_addr     : in std_logic_vector(10 downto 0);
        inst_dout     : out std_logic_vector(31 downto 0);
        data_we       : in std_logic;
        data_addr     : in std_logic_vector(10 downto 0);
        data_din      : in std_logic_vector(31 downto 0);
        data_dout     : out std_logic_vector(31 downto 0));
end RAM;

architecture syn of RAM is
    type ram_type is array (2047 downto 0) of std_logic_vector (31 downto 0);

    impure function InitRamFromFile (RamFileName : in string) return ram_type is
        FILE ramfile : text is in RamFileName;
        variable RamFileLine : line;
        variable ram : ram_type;
    begin
        for i in 0 to 1023 loop
            readline(ramfile, RamFileLine);
            read (RamFileLine, ram(i));
        end loop;
        for i in 1024 to 2047 loop
            ram(i) := x"00000000";
        end loop;
        return ram;
    end function;

    signal RAM: ram_type := InitRamFromFile("rom.data");

    begin
        process (clk)
        begin
            if clk'event and clk = '1' then
                if data_we = '1' then
                    RAM(conv_integer(data_addr)) <= data_din;
                end if;
            end if;
        end process;

        data_dout <= RAM(conv_integer(data_addr)) after 12ns;
        inst_dout <= RAM(conv_integer(inst_addr)) after 12ns;
    end syn;
```

*Εικόνα 1: Κώδικας μνήμης RAM. Το αρχείο **rom.data** θα σας δοθεί. Πάρτε ιδέες από αυτό για να δημιουργήσετε αν θέλετε και δικά σας τέτοια αρχεία.*

Σημείωση 4: Πριν συνδέσετε τον κώδικα της μνήμης στις βαθμίδες IF και MEM, επαληθεύστε (verification) πρώτα τη λειτουργία μόνη της, σ' ένα project. Φτιάξτε testbench, καλέστε ένα instance της μνήμης και δείτε πως λειτουργεί. Τι εντολές χρειάζεται να χρησιμοποιήσετε στο

testbench ώστε να διαβάσετε/γράψετε τις διευθύνσεις της για να επαληθεύσετε εξαντλητικά την ορθή λειτουργία της; Το **for-loop** της VHDL θα σας βοηθήσει.

Γ. Βαθμίδα ανάκλησης εντολών (IF)

Χρησιμοποιώντας την κατάλληλη θύρα της κύριας μνήμης και άλλη λογική, σχεδιάστε και υλοποιήστε τη βαθμίδα ανάκλησης εντολών (IF). Η βαθμίδα αποτελείται από τα παρακάτω δομικά στοιχεία:

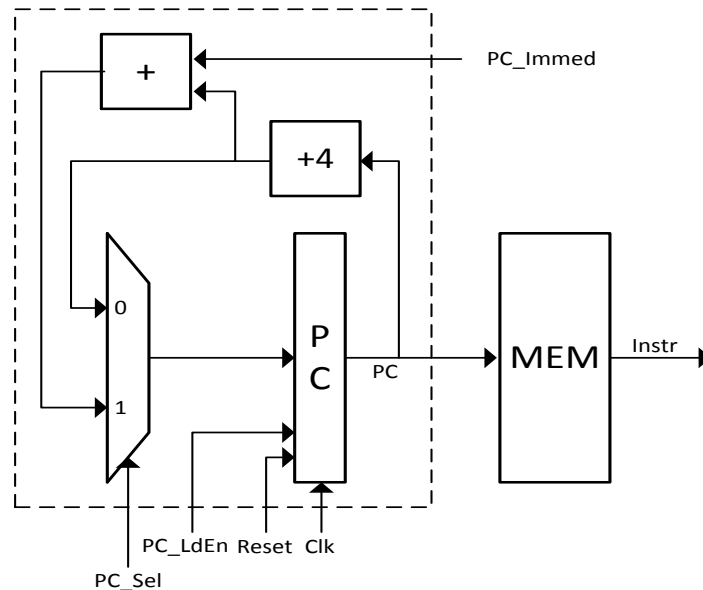
1. Τον καταχωρητή PC πλάτους 32 bits.
2. Ένα instance της μνήμης για την ανάγνωση των εντολών. Το segment τους ξεκινάει από τη διεύθυνση 0x000, οπότε δε χρειάζεται να προστεθεί κάποιο offset στη διεύθυνση από την οποία θα κάνετε την ανάγνωση. σημείωση: τα περιεχόμενα του **rom.data** αναπαριστούν εντολές.
3. Έναν αθροιστή που αυξάνει κατά 4 (incrementor), για να υπολογίζει την τιμή PC+4.
4. Έναν αθροιστή που υπολογίζει την τιμή $(PC+4) + \text{SignExt}(\text{Immed}) * 4$, για τις εντολές διακλάδωσης.
5. Ένα πολυπλέκτη 2-σε-1 που επιτρέπει να περάσει μια από τις εξής δύο τιμές για να ενημερωθεί ο PC : $(PC+4)$, ή, $(PC+4) + \text{SignExt}(\text{Immed}) * 4$.

Τα σήματα της διεπαφής της βαθμίδας IF είναι :

Σήμα	Πλάτος	Είδος	Περιγραφή
PC_Immed	32 bits	είσοδος	τιμή Immediate για εντολές b, beq, bne
PC_sel	1 bit	είσοδος	επιλογέας (<i>sel</i>) πολυπλέκτη για ενημέρωση του PC: 0 → PC+4 1 → $(PC+4) + \text{SignExt}(\text{Immed}) * 4$
PC_LdEn	1 bit	είσοδος	ενεργοποίηση εγγραφής στον PC
Reset	1 bit	είσοδος	είσοδος Reset του καταχωρητή PC
Clk	1 bit	είσοδος	ρολόι
PC	32 bits	έξοδος	διεύθυνση εντολής στην μνήμη

Εκτέλεση

1. Μελετήστε το διάγραμμα της βαθμίδας IF στην Εικόνα 2 για να κατανοήσετε τη λειτουργία της.
2. Γράψτε κώδικα VHDL που να υλοποιεί τα επιμέρους τμήματα της βαθμίδας IF και κάνετε τις κατάλληλες εσωτερικές συνδέσεις χρησιμοποιώντας την μνήμη, πολυπλέκτες, καταχωρητές και ό,τι άλλη λογική χρειάζεστε. Ονομάστε το αρχείο **IFSTAGE.vhd**
3. Γράψτε testbench και επαληθεύστε τη λειτουργία της βαθμίδας IF. Χρησιμοποιήστε το αρχείο **rom.data** για αρχικοποίηση της μνήμης, και ελέγξτε τη λειτουργία της βαθμίδας IF σε ακολουθιακές προσβάσεις αλλά και ενεργοποιώντας τις άλλες τιμές για εγγραφή στον PC.



Εικόνα 2: Σχηματικό διάγραμμα βαθμίδας ανάκλησης εντολών (IF)

Δ. Βαθμίδα αποκωδικοποίησης εντολών (DECODE)

Χρησιμοποιώντας ένα αντίγραφο (instance) του αρχείου καταχωρητών που υλοποιήσατε στην **1^η φάση** και άλλη λογική, υλοποιήστε τη βαθμίδα αποκωδικοποίησης εντολών. Η βαθμίδα αποτελείται από τα παρακάτω δομικά στοιχεία:

1. Το αρχείο καταχωρητών (RF)
2. Έναν πολυπλέκτη 2-σε-1 που επιλέγει μια από τις 2 προελεύσεις εισόδους για εγγραφή στο αρχείο καταχωρητών : ALU ή MEM δεδομένων
3. Μία μονάδα (συννεφάκι στην Εικόνα 3) που δέχεται σαν είσοδο τα 16 bits του immediate μιας εντολής και τη μετατρέπει σε ένα σήμα 32 bits, επιλέγοντας αν θα γίνει ολίσθηση του immediate, **καθώς και** αν θα γίνει zero-filling ή sign-extension του immediate προκειμένου να μετατραπεί σε 32 bit.

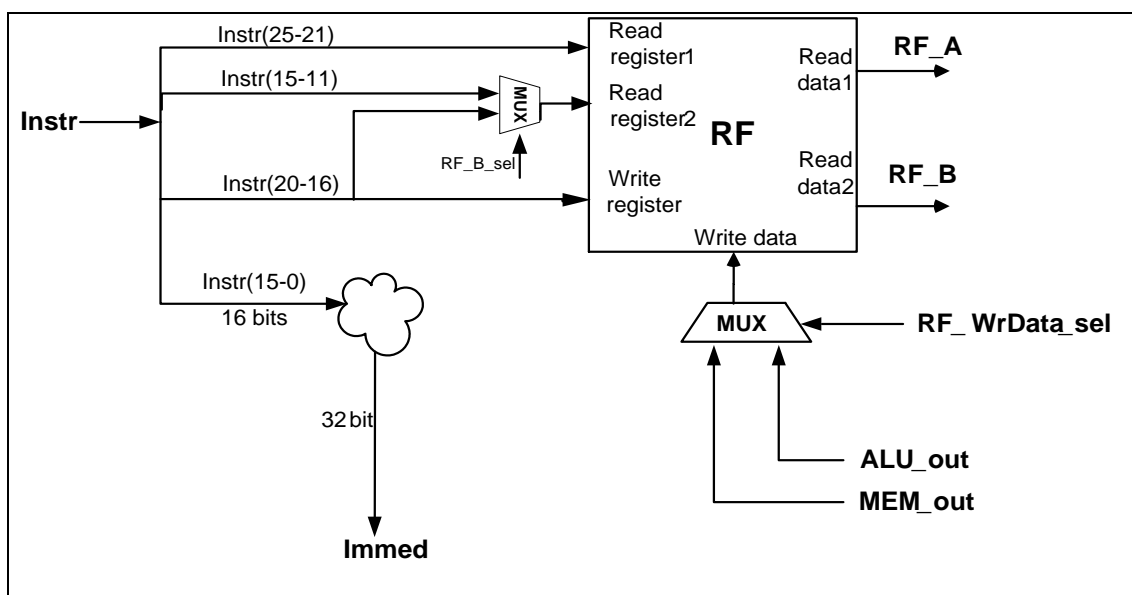
Τα σήματα της διεπαφής της βαθμίδας DEC είναι:

Σήμα	Πλάτος	Είδος	Περιγραφή
Instr	32 bit	είσοδος	η εντολή που θα αποκωδικοποιηθεί
RF_WrEn	1 bit	είσοδος	ενεργοποίηση εγγραφής καταχωρητή
ALU_out	32 bits	είσοδος	δεδομένα εγγραφής καταχωρητή προερχόμενα από την ALU
MEM_out	32 bits	είσοδος	δεδομένα εγγραφής καταχωρητή προερχόμενα από τη MEM δεδομένων
RF_WrData_sel	1 bit	είσοδος	επιλογή πεδίου που καθορίζει την προέλευση δεδομένων προς εγγραφή: 0 → ALU, 1 → MEM

Σήμα	Πλάτος	Είδος	Περιγραφή
RF_B_sel	1 bit	είσοδος	επιλογή πεδίου που καθορίζει τον δεύτερο καταχωρητή ανάγνωσης: 0 → Instr(15-11), 1 → Instr(20-16)
ImmExt	2 bits	είσοδος	zero-filling , sign-extension , ολίσθηση
Clk	1 bit	είσοδος	ρολόι
Immed	32 bits	έξοδος	Immediate προς τις επόμενες βαθμίδες
RF_A	32 bits	έξοδος	τιμή του 1 ^{ου} καταχωρητή
RF_B	32 bits	έξοδος	τιμή του 2 ^{ου} καταχωρητή

Εκτέλεση

1. Μελετήστε το διάγραμμα της βαθμίδας DEC στην Εικόνα 3 για να κατανοήσετε τη λειτουργία της.
2. Ο καταχωρητής R0 θα έχει πάντα την τιμή "0" (μηδέν).
3. Γράψτε κώδικα VHDL που να υλοποιεί τα επιμέρους τμήματα της βαθμίδας DEC και κάνετε τις κατάλληλες εσωτερικές συνδέσεις χρησιμοποιώντας το Register File που παράγατε στην 1^η φάση, πολυπλέκτες, καταχωρητές και ό,τι άλλη λογική χρειάζεστε. Ονομάστε το αρχείο **DECSTAGE.vhd**
4. Γράψτε testbench και επαληθεύστε τη λειτουργία της βαθμίδας DEC. Προσομοιώστε καλύπτοντας τις βασικές κατηγορίες εντολών ώστε να επιβεβαιώσετε τη λογική αποκωδικοποίησης. Στην προσομοίωση καλύψτε όσο το δυνατόν περισσότερες περιπτώσεις.
5. Προσοχή: από το opcode της εντολής καταλαβαίνουμε αν χρειάζεται sign-extend, ή zero-fill, και αν χρειάζεται ολίσθηση ή όχι. Το "συννεφάκι" κάνει όλες αυτές τις περιπτώσεις που είναι 4 σύνολο, και γι' αυτό χρειάζονται 2 bits για το σήμα ελέγχου ImmExt.



Εικόνα 3: Σχηματικό διάγραμμα βαθμίδας αποκωδικοποίησης εντολών (DEC)

Ε. Βαθμίδα εκτέλεσης εντολών (EX)

Χρησιμοποιώντας την ALU που σχεδιάσατε στην 1^η φάση και άλλη λογική, υλοποιήστε τη βαθμίδα εκτέλεσης αριθμητικών και λογικών εντολών. Η βαθμίδα αποτελείται από τα παρακάτω δομικά στοιχεία:

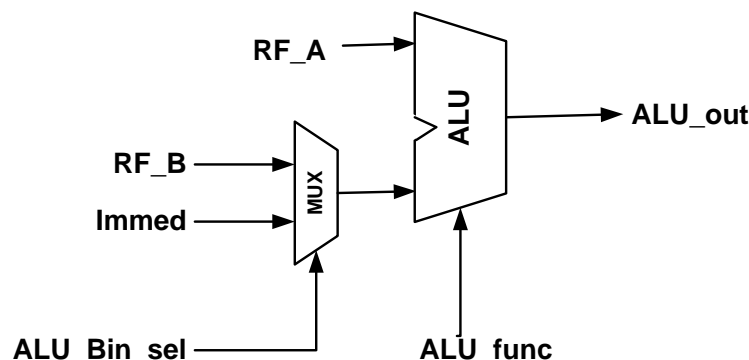
1. Την ALU
2. Πολυπλέκτη που επιλέγει ποιος είναι ο δεύτερος τελεστέος της ALU.

Τα σήματα της διεπαφής της βαθμίδας EX είναι:

Σήμα	Πλάτος	Είδος	Περιγραφή
RF_A	32 bits	είσοδος	RF[rs]
RF_B	32 bits	είσοδος	RF[rt] ή RF[rd]
Immed	32 bits	είσοδος	Immediate
ALU_Bin_sel	1 bit	είσοδος	επιλογή Εισόδου B της ALU από RF_B ή Immediate 1 → Immed 0 → RF_B
ALU_func	4 bit	είσοδος	πράξη ALU
ALU_out	32 bits	έξοδος	αποτέλεσμα ALU
ALU_zero	1 bit	έξοδος	zero flag

Εκτέλεση

1. Μελετήστε το διάγραμμα της βαθμίδας EX στην Εικόνα 4 για να κατανοήσετε τη λειτουργία της.
2. Γράψτε κώδικα VHDL που να υλοποιεί τα επιμέρους τμήματα της βαθμίδας EX και κάνετε τις κατάλληλες εσωτερικές συνδέσεις χρησιμοποιώντας πολυπλέκτες, καταχωρητές, και ότι άλλη λογική χρειάζεστε. Ονομάστε το αρχείο **EXSTAGE.vhd**
3. Γράψτε testbench και επαληθεύστε τη λειτουργία της βαθμίδας EX. Καλύψτε όσες περισσότερες περιπτώσεις γίνεται.



Εικόνα 4: Σχηματικό διάγραμμα βαθμίδας εκτέλεσης εντολών (EX)

ΣΤ. Βαθμίδα πρόσβασης μνήμης (MEM)

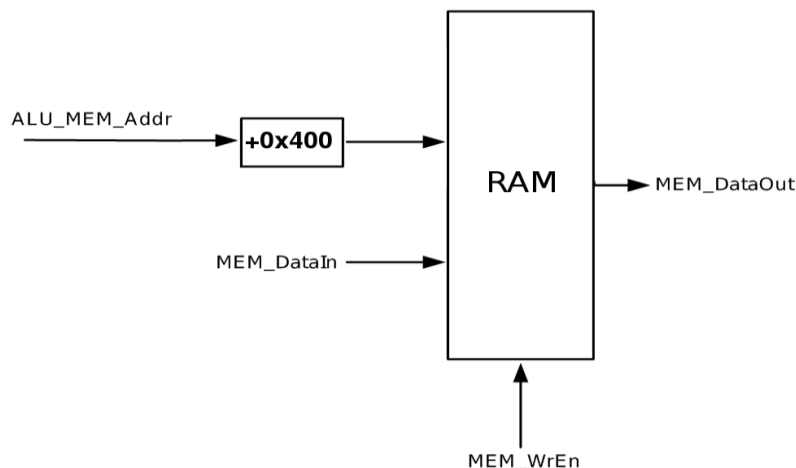
Χρησιμοποιήστε ένα instance της μνήμης της Εικόνας 1. Επειδή η μνήμη είναι κοινή για τα δεδομένα και τις εντολές (αυτό θα γίνει σε επόμενη φάση), ο επεξεργαστής θα πρέπει να διαβάζει και να γράφει τα δεδομένα από το σωστό τμήμα (text segment για εντολές, και data segment για δεδομένα). Για το λόγο αυτό, η διεύθυνση της μνήμης που θα διαβάζεται πρέπει να είναι η διεύθυνση που στέλνει η ALU **προστιθέμενη κατά 0x400 (offset)**.

Τα σήματα της διεπαφής της βαθμίδας MEM είναι:

Σήμα	Πλάτος	Είδος	Περιγραφή
clk	1 bit	είσοδος	Ρολόι
Mem_WrEn	1 bit	είσοδος	σημαία (flag) ενεργοποίησης εγγραφής στη μνήμη
ALU_MEM_Addr	32 bits	είσοδος	αποτέλεσμα ALU (βλέπε εντολές lb, sb, lw, sw) (από EX)
MEM_DataIn	32 bits	είσοδος	αποτέλεσμα RF[rd] για αποθήκευση στη μνήμη για εντολές swar και sb, sw (από EX)
MEM_DataOut	32 bits	έξοδος	δεδομένα που φορτώθηκαν από τη μνήμη προς εγγραφή σε καταχωρητή για εντολές lb, lw (προς RF)
MM_Addr	32 bits	έξοδος	διεύθυνση προς module μνήμης
MM_Wr	32 bits	έξοδος	σήμα ενεργοποίησης εγγραφής προς module μνήμης
MM_WrData	32 bits	έξοδος	δεδομένα για εγγραφή προς module μνήμης
MM_RdData	32 bits	είσοδος	δεδομένα που αναγνώστηκαν από το module μνήμης

Εκτέλεση

1. Μελετήστε το διάγραμμα της βαθμίδας MEM στην Εικόνα 5 για να κατανοήσετε τη λειτουργία της.
2. Γράψτε κώδικα VHDL που να υλοποιεί τα επιμέρους τμήματα της βαθμίδας MEM και συνδέστε με την εξωτερική μνήμη της Εικόνας 1. Ονομάστε το αρχείο **MEMSTAGE.vhd**
3. Γράψτε testbench και επαληθεύστε τη λειτουργία της βαθμίδας MEM.



Εικόνα 5: Σχηματικό διάγραμμα βαθμίδας πρόσβασης στη μνήμη (MEM)

Παραδοτέα

1. Αρχεία κώδικα VHDL, με τα testbenches
2. Waveform configuration files (.wcfg)

Διευκρίνιση σχετικά με τον αριθμό των bits διεύθυνσης μνήμης

Ο PC και οι διευθύνσεις δεδομένων που παράγουν οι εντολές lw/sw είναι 32 bits και αναφέρονται σε bytes. Η μνήμη είναι οργανωμένη σε λέξεις (1 word=4 bytes) και δέχεται διεύθυνση 11 bits.

Η μετατροπή από διεύθυνση 32-bit σε διεύθυνση 11-bit γίνεται με τον ακόλουθο τρόπο:

32-bits διεύθυνση byte	11-bits διεύθυνση μνήμης
00....00000000 (έως 00..00000011)	00000000000
00....00000100	00000000001
00....00001000	00000000010
00....00001100	00000000011

Οπότε, από την διεύθυνση 32 bit address[31..0] **επιλέγουμε τα bits [12..2], αγνοώντας έτσι τα 2 λιγότερο σημαντικά bits**, τα οποία ουσιαστικά **αναφέρονται στο byte μέσα στην λέξη**.