



ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
ΗΡΥ 302: ΟΡΓΑΝΩΣΗ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΑΡΙΝΟ ΕΞΑΜΗΝΟ 2021-22

Παραεμβάσεις: Κυπριανός Παπαδημητρίου

Εργασία #1: Σχεδίαση επεξεργαστή ενός κύκλου
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 όλη 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), δομική (structural), dataflow, καθώς και του περιβάλλοντος της Xilinx. Γνώσεις που αποκτήθηκαν στην Προχωρημένη Λογική Σχεδίαση, στη Λογική Σχεδίαση, και στους Ψηφιακούς Υπολογιστές.

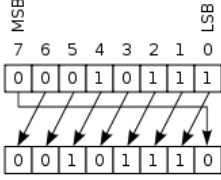
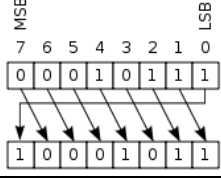
Διεξαγωγή

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)
Ovf	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 = 0101	Λογικό “NAND”	Out = ! (A & B)
Op = 0110	Λογικό “NOR”	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	

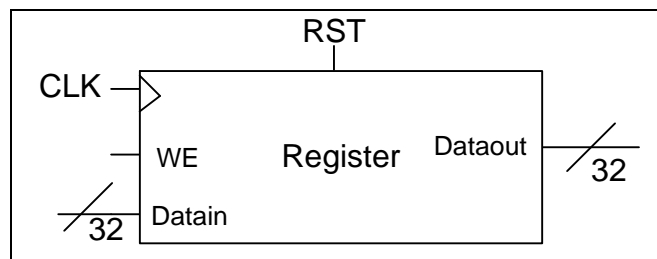
- Υλοποιήστε την ALU σε VHDL. Χρησιμοποιήστε Behavioral κώδικα όσο περισσότερο μπορείτε. Για παράδειγμα, για να φτιάξετε έναν αθροιστή χρησιμοποιήστε τον τελεστή «+» και για την αφαίρεση «-». Μην σχεδιάσετε κυκλώματα αθροιστών ξεκινώντας με τις βασικές λογικές πύλες, π.χ. μην φτιάξετε half-adder με XOR, 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**, **Cout** 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 με χρήση του **after** που περιγράφηκε παραπάνω.



Εικόνα 1: Καταχωρητής πλάτους 32 bits

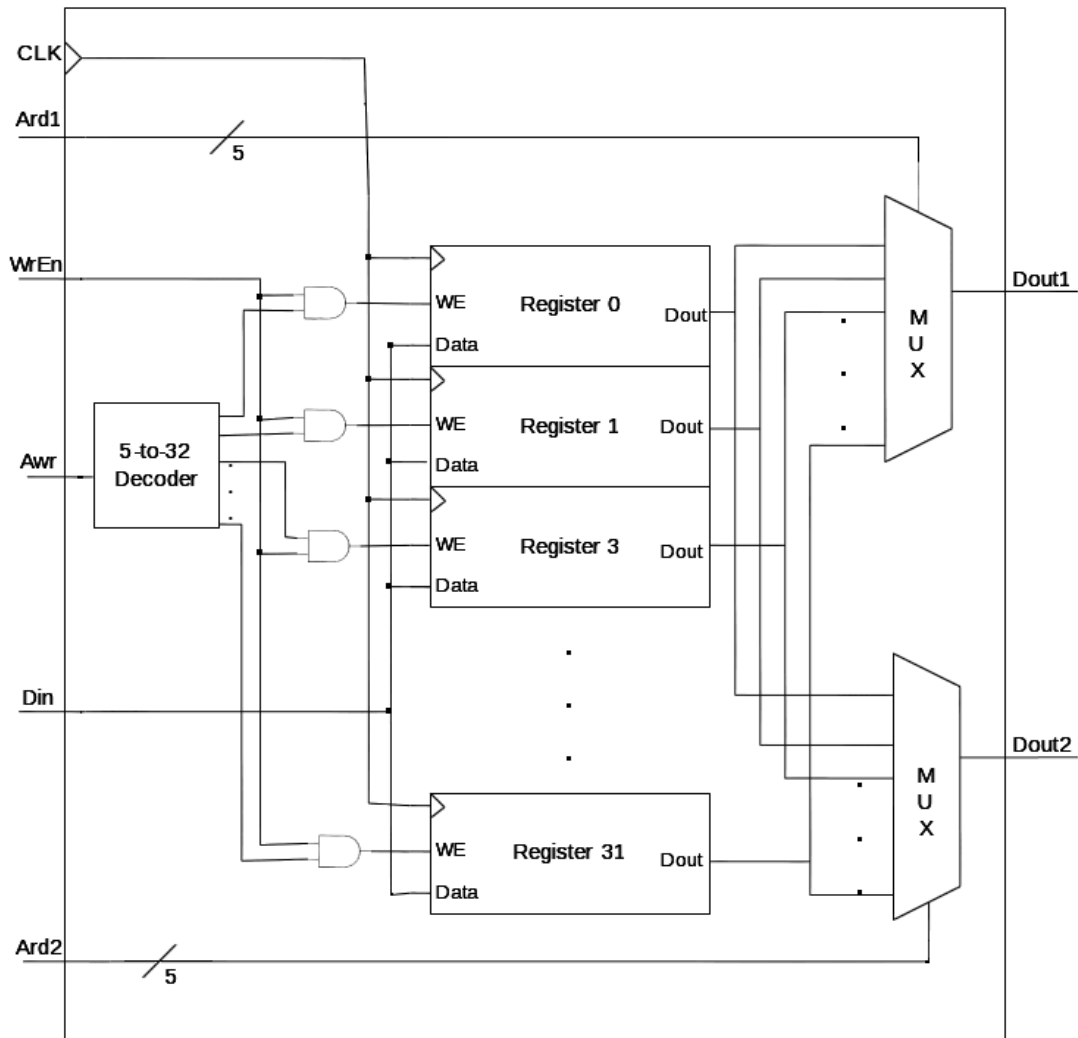
B2. Δημιουργία αρχείου καταχωρητών (RF)

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
Rst	Είσοδος (1 bit)	Reset (προαιρετικό - διερευνήστε αν απαιτείται ή όχι)

- Υλοποιήστε σε VHDL το αρχείο καταχωρητών χρησιμοποιώντας τον καταχωρητή που υλοποιήσατε στο B1.
- 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.
- Τα κυκλώματα που απεικονίζονται στην Εικόνα 2 είναι συνδυαστικά, εκτός από τους registers.
- 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 χρησιμοποιείστε 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. Την περίοδο αυτή θα χρησιμοποιήσετε και για τον επεξεργαστή σας.



Εικόνα 2: Αρχείο καταχωρητών (RF)

Γενικές οδηγίες – αφορούν όλες τις φάσεις της εργασίας

- Αν πρόκειται να χρησιμοποιήσετε δικό σας υπολογιστή μόνο, και κατά την προσέλευσή σας στο εργαστήριο π.χ. στο laptop σας, μπορείτε να δουλέψετε αποκλειστικά με το Xilinx Vivado. Στους υπολογιστές του εργαστηρίου διατίθεται η Xilinx ISE 14.7 και του μηχανογραφικού η Xilinx ISE 13.4. Οπότε 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.
- Μαζί με τους κώδικες VHDL υποβάλλετε 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 14.7 ή στο 13.4. Προσοχή όμως, θα πρέπει να εξοικειωθείτε με το ISE 14.7 ή το 13.4. Δεν είναι δύσκολο μελετήστε το εγχειρίδιο του Xilinx ISE και τις συχνές ερωτήσεις.
- 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. 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 του.
- Γράψτε "καθαρό κώδικα" VHDL με τα modules ιεραρχικά οργανωμένα, π.χ. topModule, πολυπλέκτες, αποκωδικοποιητές, 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 σχεδίασή σας.

- Όταν γράφετε κώδικα με **if..then** διερευνήστε αν είναι απαραίτητο να κλείσετε όλα τα **else**. Μπορεί να είναι απαραίτητο, μπορεί και όχι. Το ίδιο και για το **others**.
- 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 αποκωδικοποιητής, 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) ή στο '0' (active low), ανάλογα και με την υπόλοιπη σχεδίαση.
- 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 αποκωδικοποιητή, συγκριτή, πολυπλέκτη, register κλπ. Δηλαδή βάζετε **10ns** καθυστέρηση μόνο στην τελική έξοδο του module. Ανάλογα το υποκύκλωμα μπορείτε να βάλετε και μικρότερη καθυστέρηση, π.χ. **after 5ns**.
- 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 εκάστοτε 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 αποκωδικοποιητή, 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 παρακολουθήσετε στην προσομοίωση. Φτιάξτε testbench, και εκεί κάνετε instance το topLevel.
- "**for-loop**" statement της VHDL. Μην το χρησιμοποιείτε για να περιγράφετε κυκλώματα, π.χ. έναν συγκριτή, πολυπλέκτη, αποκωδικοποιητή, μετρητή, FSM κλπ. Να το χρησιμοποιείτε μόνο σε testbench, πχ. να παράγετε με εύκολο τρόπο όλους τους διαφορετικούς συνδυασμούς τιμών που θα μπορούσαν να εμφανιστούν στις 2 εισόδους ενός κυκλώματος συγκριτή ισότητας.

2^η φάση: χρόνος ολοκλήρωσης 12 ημέρες

«Σχεδίαση των βασικών βαθμίδων του datapath ενός απλού επεξεργαστή»

- Μελετήστε πρώτα καλά όλη 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 DEC)
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-άδες 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	and	$RF[rd] \leftarrow RF[rs] \text{ AND } RF[rt]$
100000	110011	or	$RF[rd] \leftarrow RF[rs] \text{ } RF[rt]$
100000	110100	not	$RF[rd] \leftarrow \neg RF[rs]$
100000	110101	nand	$RF[rd] \leftarrow RF[rs] \text{ NAND } RF[rt]$
100000	110110	nor	$RF[rd] \leftarrow RF[rs] \text{ NOR } RF[rt]$
100000	111000	sra	$RF[rd] \leftarrow RF[rs] \gg 1$
100000	111001	srl	$RF[rd] \leftarrow RF[rs] \gg 1$ (Logical, zero fill MSB)
100000	111010	sll	$RF[rd] \leftarrow RF[rs] \ll 1$ (Logical, zero fill LSB)
100000	111100	rol	$RF[rd] \leftarrow \text{Rotate left}(RF[rs])$
100000	111101	ror	$RF[rd] \leftarrow \text{Rotate right}(RF[rs])$
111000	-	li	$RF[rd] \leftarrow \text{SignExtend}(\text{Imm})$
111001	-	lui	$RF[rd] \leftarrow \text{Imm} \ll 16$ (zero-fill)
110000	-	addi	$RF[rd] \leftarrow RF[rs] + \text{SignExtend}(\text{Imm})$
110010	-	nandi	$RF[rd] \leftarrow RF[rs] \text{ NAND } \text{ZeroFill}(\text{Imm})$
110011	-	ori	$RF[rd] \leftarrow RF[rs] \text{ } \text{ZeroFill}(\text{Imm})$
111111	-	b	$PC \leftarrow PC + 4 + (\text{SignExtend}(\text{Imm}) \ll 2)$
000000	-	beq	if ($RF[rs] == RF[rd]$) $PC \leftarrow PC + 4 + (\text{SignExtend}(\text{Imm}) \ll 2)$ else $PC \leftarrow PC + 4$
000001	-	bne	if ($RF[rs] != RF[rd]$) $PC \leftarrow PC + 4 + (\text{SignExtend}(\text{Imm}) \ll 2)$ else $PC \leftarrow PC + 4$
000011	-	lb	$RF[rd] \leftarrow \text{ZeroFill}(31 \text{ downto } 8) \& \text{MEM}[RF[rs] + \text{SignExtend}(\text{Imm})](7 \text{ downto } 0)$
000111	-	sb	$\text{MEM}[RF[rs] + \text{SignExtend}(\text{Imm})] \leftarrow \text{ZeroFill}(31 \text{ downto } 8) \& RF[rd]$ (7 downto 0)
001111	-	lw	$RF[rd] \leftarrow \text{MEM}[RF[rs] + \text{SignExtend}(\text{Imm})]$
011111	-	sw	$\text{MEM}[RF[rs] + \text{SignExtend}(\text{Imm})] \leftarrow RF[rd]$

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 μια διεύθυνση **αρκετά παρακάτω στη μνήμη**, η οποία προκύπτει από πρόσθεση και επεξεργασία με τον αριθμό **0x400 (offset)**. Για περισσότερες πληροφορίες σχετικά με την κατάτμηση της μνήμης ανατρέξτε σε ύλη μαθήματος προηγούμενου εξαμήνου.

Χρησιμοποιήστε τον κώδικα της Εικόνας 3 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 3 που επισημαίνεται με έντονα γράμματα. Ο συγκεκριμένος κώδικας “διαβάζει” το αρχείο **rom.data** και φορτώνει κάθε γραμμή του σε μια διαφορετική διεύθυνση της μνήμης. Το rom.data είναι ουσιαστικά ένα πρόγραμμα με εντολές ASSEMBLY όπου σε κάθε γραμμή βρίσκεται μια εντολή κωδικοποιημένη σε δυαδική μορφή 32-bits. 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, δεν 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**. Κάνετε ξεχωριστό testbench για την κάθε περίπτωση ώστε να επαληθεύσετε ότι κάνατε σωστά τις συνδέσεις. Στο κάθε testbench βάλετε ένα instance της μνήμης,

με **component/port map**. Στην επόμενη φάση της εργασίας θα έχουμε 1 μόνο instance της μνήμης συνδεδεμένο και με τις δυο βαθμίδες, όπου πλέον δεν θα υπάρχουν αχρησιμοποίητες είσοδοι και έξοδοι.

```
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 3: Κώδικας μνήμης RAM. Το αρχείο **rom.data** θα σας δοθεί. Πάρτε ιδέες από αυτό για να δημιουργήσετε 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:

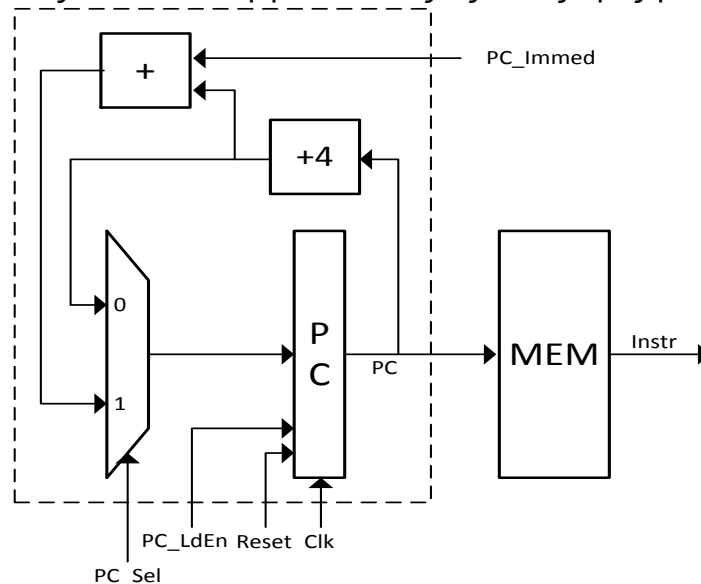
1. Τον καταχωρητή PC πλάτους 32 bits.
2. Έναν αθροιστή που αυξάνει κατά 4 (incrementor), για να υπολογίζει την τιμή PC+4.
3. Έναν αθροιστή που υπολογίζει την τιμή $(PC+4) + \text{SignExt}(\text{Immed}) * 4$, για τις εντολές διακλάδωσης.
4. Ένα πολυπλέκτη 2-σε-1 που επιτρέπει να περάσει μια από τις εξής 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 (sel) 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 4 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 να φορτώσετε τη μνήμη εντολών, και ελέγξτε τη λειτουργία της βαθμίδας IF σε ακολουθιακές προσβάσεις αλλά και ενεργοποιώντας τις άλλες τιμές για εγγραφή στον PC.



Εικόνα 4: Σχηματικό διάγραμμα βαθμίδας ανάκλησης εντολών (IF)

Σημείωση: Το instance της μνήμης της Εικόνας 4 βρίσκεται εκτός του **IFSTAGE.vhd** και χρησιμοποιείται 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. Τα περιεχόμενα του **rom.data** αναπαριστούν εντολές.

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 5) 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, αν θα γίνει zero-filling, ή, sign-extension του immediate προκειμένου να μετατραπεί σε 32 bit.

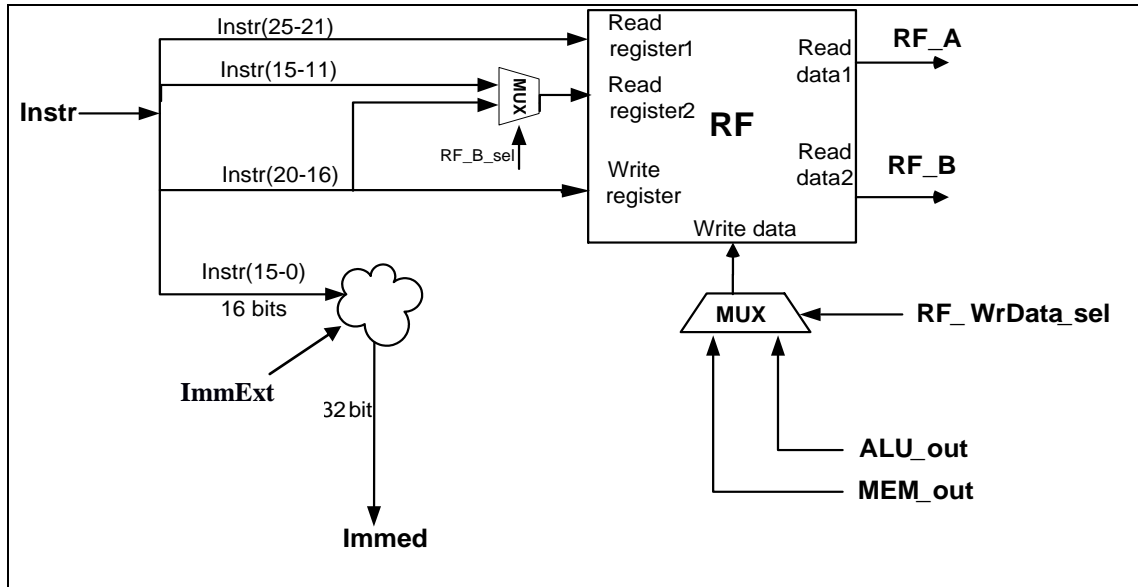
Τα σήματα της διεπαφής της βαθμίδας DEC είναι:

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
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	ανάλογα της τιμής του σήματος ImmExt θα γίνει: zero-filling , sign-extension , ολίσθηση. <u>Σημ.:</u> ορίστε εσείς ποια τιμή του ImmExt αντιστοιχεί σε ποια ενέργεια
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 5 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 5: 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:

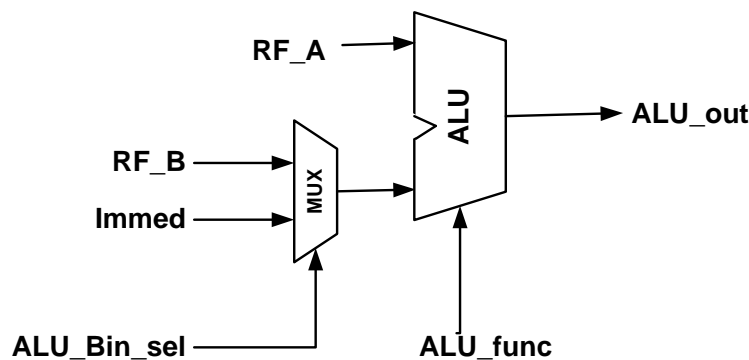
1. Την ALU
2. Πολυπλέκτη που επιλέγει ποιος είναι ο δεύτερος τελεστέος της ALU.

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.



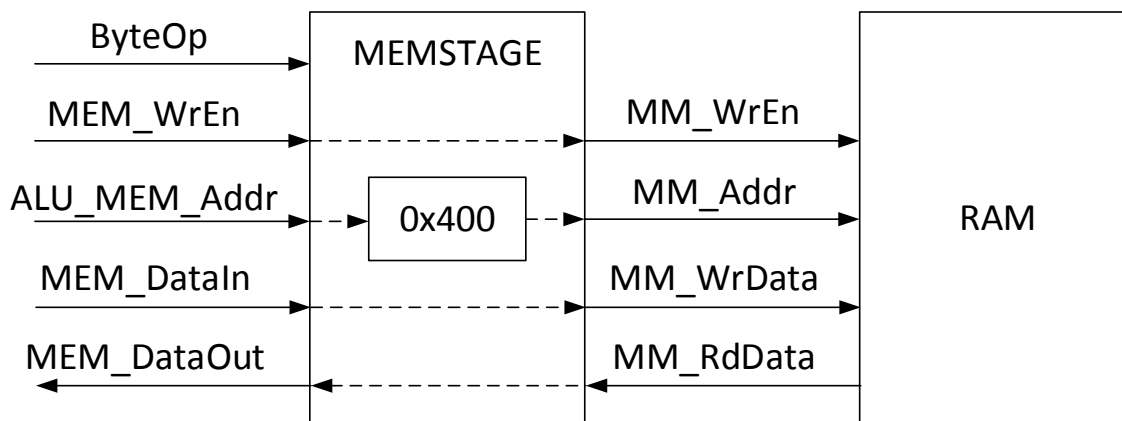
Εικόνα 6: Σχηματικό διάγραμμα βαθμίδας εκτέλεσης εντολών (EX)

ΣΤ. Βαθμίδα πρόσβασης μνήμης (MEM)

Χρησιμοποιήστε ένα instance της μνήμης της Εικόνας 3. Σε επόμενη φάση θα έχετε μια μνήμη μόνο, κοινή για δεδομένα και εντολές, οπότε ο επεξεργαστής θα πρέπει να διαβάζει και να γράφει τα δεδομένα από το σωστό τμήμα (TEXT segment για να διαβάζει εντολές, και DATA segment για να γράφει/διαβάζει 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
ByteOp	1 bit	είσοδος	σήμα ελέγχου για επιλογή <i>lw/sw</i> ('0') ή <i>lb/sb</i> ('1')
Mem_WrEn	1 bit	είσοδος	σημαία (flag) ενεργοποίησης εγγραφής στη μνήμη
ALU_MEM_Addr	32 bits	είσοδος	αποτέλεσμα ALU (βλέπε εντολές <i>lb, sb, lw, sw</i>) (από EX)
MEM_DataIn	32 bits	είσοδος	αποτέλεσμα RF[rd] για αποθήκευση στη μνήμη για εντολές <i>swar</i> και <i>sb, sw</i> (από EX)
MEM_DataOut	32 bits	έξοδος	δεδομένα που φορτώθηκαν από τη μνήμη προς εγγραφή σε καταχωρητή για εντολές <i>lb, lw</i> (προς RF)
MM_WrEn	1 bit	έξοδος	σήμα ενεργοποίησης εγγραφής προς module μνήμης
MM_Addr	32 bits	έξοδος	διεύθυνση 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



Εικόνα 7: Σχηματικό διάγραμμα βαθμίδας πρόσβασης στη μνήμη (MEM). Προσοχή: σήμα εισόδου CLK παίρνει μόνο η RAM, όχι το MEMSTAGE. Επίσης, για την είσοδο MM_Addr της μνήμης, δεν σας χρειάζονται 32 bits (πόσα σας χρειάζονται;).

Διευκρίνιση: οι διακεκομμένες γραμμές εντός του MEMSTAGE module της Εικόνας 7 δεν σημαίνουν hardwiring των εισόδων στις εξόδους. Είναι ενδεικτικές, και υποδηλώνουν τη συσχέτιση μεταξύ των σημάτων εισόδου/εξόδου. Το MEMSTAGE περιλαμβάνει και επιπλέον λογική εκτός του adder (+0x400) που φαίνεται στην εικόνα.

Εκτέλεση

1. Μελετήστε το διάγραμμα της βαθμίδας MEM στην Εικόνα 7 για να κατανοήσετε τη λειτουργία της. Τι άλλα κυκλώματα θα χρειαστείτε εσωτερικά σε αυτή τη βαθμίδα;
2. Γράψτε κώδικα VHDL που να υλοποιεί τα επιμέρους τμήματα της βαθμίδας MEM και συνδέστε με την εξωτερική μνήμη της Εικόνας 3. Ονομάστε το αρχείο **MEMSTAGE.vhd**
3. Γράψτε testbench και επαληθεύστε τη λειτουργία της βαθμίδας MEM.

Διευκρίνιση σχετικά με τον αριθμό των bits διεύθυνσης μνήμης

Ο PC καθώς και οι διευθύνσεις δεδομένων που παράγουν οι εντολές lw/sw είναι 32 bits και αναφέρονται σε bytes. Η μνήμη είναι οργανωμένη σε “λέξεις” (δηλαδή “words” όπου 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 [31..0]	11-bits διεύθυνση μνήμης [12..2]
00....00000000 (έως 00..00000011)	00000000000
00....00000100 (έως 00..00000111)	00000000001
00....00001000 (έως 00..00001011)	00000000010
00....00001100 (έως 00..00001111)	00000000011
...	...

Όπως φαίνεται στον παραπάνω πίνακα από τα [31..0] επιλέγουμε τα bits [12..2] για να διευθυνσιοδοτήσουμε τη μνήμη, αγνοώντας έτσι τα 2 λιγότερο σημαντικά bits που αναφέρονται στο byte μέσα στην λέξη. Οπότε, αν και ο PC αυξάνεται κατά 4 (PC+4), επειδή εμείς επιλέγουμε τα bits [12..2] η διεύθυνση της μνήμης θα αυξάνεται κατά ένα (1) ως εξής: 0, 1, 2, 3, 4, 5, 6 κοκ.

3^η φάση: χρόνος ολοκλήρωσης 10 ημέρες

«Επεξεργαστής : ολοκλήρωση του datapath, δημιουργία του control, και σύνδεση datapath-control»

- Μελετήστε πρώτα καλά όλη την εκφώνηση -

Σκοπός της 3^{ης} φάσης

Είναι η ολοκλήρωση και ο έλεγχος του datapath, η δημιουργία του control που θα παράγει τα σήματα ελέγχου, και η σύνδεση/επικοινωνία των δύο (datapath+control) για την ολοκλήρωση της δημιουργίας του επεξεργαστή ενός κύκλου.

Εκτέλεση

1. Γράψτε τον κώδικα VHDL που υλοποιεί το datapath του επεξεργαστή (**DATAPATH.vhd**) στο οποίο θα κάνετε τη σύνδεση των βαθμίδων IF, DECODE, EX και MEM που σχεδιάσατε στην προηγούμενη φάση. Συλλέξτε όλα τα σήματα ελέγχου. Η διεπαφή του DATAPATH.vhd περιλαμβάνει όλα τα σήματα ελέγχου όλων των επιμέρους βαθμίδων που datapath.
2. Η μνήμη MEM της εικόνας 4 και η μνήμη RAM της εικόνας 7 αναφέρονται πλέον σε ένα κοινό module μνήμης. Θα υπάρχει λοιπόν **μια κοινή μνήμη RAM** (όχι εντός του DATAPATH.vhd) η οποία θα είναι συνδεδεμένη με τις βαθμίδες IF και MEM. Κατά συνέπεια η διεπαφή του datapath θα πρέπει να μπορεί με τα κατάλληλα σήματα να εξυπηρετήσει την απαραίτητη επικοινωνία των βαθμίδων IF και MEM με την μνήμη.
3. Δημιουργήστε προγράμματα αναφοράς (δηλαδή δικά σας αρχεία **rom.data** με εντολές) και τα κατάλληλα testbenches που θα παράγουν τα σήματα ελέγχου για να επιβεβαιώσετε την ορθή λειτουργία του datapath. Το module μνήμης μπορείτε να το κάνετε **port map** στο κάθε testbench.
4. Χρησιμοποιήστε τα προγράμματα αναφοράς #1 και #2 που δίνονται σε παρακάτω σελίδα (κωδικοποιήστε τις εντολές σε binary για να φτιάξετε τα αντίστοιχα αρχεία **rom.data**), για να επαληθεύσετε τη λειτουργία του επεξεργαστή.
5. Γράψτε τον κώδικα VHDL που υλοποιεί το control του επεξεργαστή (**CONTROL.vhd**) το οποίο θα παράγει τα σωστά σήματα ελέγχου για την κάθε εντολή.
6. Δημιουργήστε ένα testbench για το control για να επιβεβαιώσετε την ορθή λειτουργία του. Στο συγκεκριμένο testbench δώστε όλες τις εντολές που υποστηρίζει ο επεξεργαστής σας με τη σειρά που δίνονται στον πίνακα της σελίδας 6.
7. Γράψτε τον κώδικα VHDL που υλοποιεί το topLevel του επεξεργαστή ενός κύκλου (**PROC_SC.vhd**). Το συγκεκριμένο αρχείο περιλαμβάνει τρία components (**port map**): datapath, control και το κοινό module μνήμης.
8. Για να ελέγξετε το κύκλωμά σας δώστε στο testbench **Reset='1'** για λίγους κύκλους και μετά **Reset='0'** έτσι ώστε να ξεκινήσει η εκτέλεση από την θέση 0.

3^η φάση - λίστα σημάτων παρακολούθησης στην προσομοίωση

Η παρακάτω λίστα θα σας βοηθήσει στον έλεγχο, και στην αποτελεσματική παρακολούθηση της προσομοίωσης. Για τις κυματομορφές της 3ης φάσης ετοιμάστε αρχείο `.wcfg` με τα παρακάτω τουλάχιστον σήματα, με τη σειρά που δίνονται:

- Reset
- Clk
- Instruction
- PC
- PC_LdEn
- RF_WrEn
- RF_ADDRESS_WRITE
- RF_WrData_Sel
- ALU_func
- ALU_overflow
- ALU_zero
- MEM_WrEn
- καταχωρητές r3, r5, r10, r16
- 5 θέσεις μνήμης που να αφορούν το DATA segment, δηλ. ξεκινώντας από την διεύθυνση byte 0x400 (1024 σε δεκαδικό). σημείωση: Η πρόσθεση με το 0x400 γίνεται **πριν την επιλογή των bits [12..2]**, οπότε αυτές αντιστοιχούν στις λέξεις (θέσεις) 256-260 της μνήμης. Αν την πρόσθεση με το 0x400 την κάνετε **μετά την επιλογή των bits [12..2]**, προσαρμόστε κατάλληλα τις λέξεις (θέσεις) που θα δείξετε.

Καλώδια/τιμές με πολλά bits δείξτε τα σε HEX ή DEC αν το θεωρείτε πιο κατάλληλο, π.χ. βολεύει στην τιμή του PC, στο αποτέλεσμα πράξης, στη διεύθυνση μνήμης κλπ.

Πρόγραμμα αναφοράς #1

```
00: addi r5, r0, 8
04: ori  r3, r0, 0xABCD
08: sw   r3, 4(r0)      // γράφει στην διεύθυνση 0x4 => 0x404 την τιμή 0x0000ABCD
0C: lw   r10, -4(r5)     // διαβάζει από την διεύθυνση 0x4 => 0x404 την τιμή 0x0000ABCD
10: lb   r16, 4(r0)      // διαβάζει byte από την διεύθυνση 0x4 => 0x404 την τιμή 0x000000CD
14: nand r4, r10, r16
```

Εντολή	Opcode	rs	rd	rt	func	Immed	Binary	Hex
00: addi r5,r0,8	110000	00000	00101	-	-	00000000000001000	1100 00 00 000 0 0101 0000 0000 0000 1000	C005 0008
04: ori r3,r0,ABCD	110011	00000	00011	-	-	10101011111001101	1100 11 00 000 0 0011 1010 1011 1100 1101	CC03 ABCD
08: sw r3,4(r0)	011111	00000	00011	-	-	00000000000000100	0111 11 00 000 0 0011 0000 0000 0000 0100	7C03 0004
0C: lw r10,-4(r5)	001111	00101	01010	-	-	11111111111111100	0011 11 00 101 0 1010 1111 1111 1111 1100	3CAA FFFC
10: lb r16,4(r0)	000011	00000	10000	-	-	00000000000000100	0000 11 00 000 1 0000 0000 0000 0000 0100	0C10 0004
14: nand r4,r10,r16	100000	01010	00100	10000	110101	-	1000 00 01 010 0 0100 1000 0 000 00 11 0101	8144 8035

Πρόγραμμα αναφοράς #2

```
// *** Δεύτερο πρόγραμμα, μόνο διακλαδώσεις, εκτελεί δύο εντολές με αποτυχημένη διακλάδωση και ξανά
00: bne r5, r5, 8      // αποτυχημένη διακλάδωση
04: b -2               // branch (PC=04 + 4 -2*4 = 00) infinite loop!
08: addi r1, r0, 1     // δεν θα εκτελεστεί
```

Εντολή	Opcode	rs	rd	rt	func	Immed	Binary	Hex
00: bne r5,r5,8	000001	00101	00101	-	-	00000000000001000	0000 01 00 101 0 0101 0000 0000 0000 1000	04A5 0008
04: b -2	111111	00000	00000	-	-	11111111111111110	1111 11 00 000 0 0000 1111 1111 1111 1110	FC00 FFFE
08: addi r1,r0,1	110000	00000	00001	-	-	00000000000000001	1100 00 00 000 0 0001 0000 0 000 00 00 0001	C001 0001

Αναλυτική περιγραφή παραδοτέων Εργασίας #1

- Ακολουθήστε τις οδηγίες όπως είναι ακριβώς -

Ο τελικός φάκελος που θα υποβάλλετε θα φέρει το επίθετο και τον αριθμό μητρώου σας π.χ. [a_christodoulou_2019123456](#). Μην ξεχάσετε το “a_”. **Κάνετε zip τον φάκελο (συγκεκριμένα zip, όχι rar ή 7z ή οτιδήποτε άλλο)**. Προσοχή: το επίθετο σας να είναι σε greeklish και επίσης παρεμβάλλετε το σύμβολο “_” μεταξύ του επιθέτου και του ΑΜ. Εσωτερικά δημιουργείτε 3 υποφακέλους: **REPORT**, **SOURCES**, **WAVEFORMS**, και οι 3 με κεφαλαία γράμματα. Προσέξτε την ονομασία του φακέλου να είναι σωστή και πλήρης. Αν είναι λάθος ο κώδικάς σας δεν θα βαθμολογηθεί.

1. Αναφορά (μέχρι 6 σελίδες, PDF)

Αποθηκεύστε την στον υποφάκελο REPORT. Η αναφορά δεν θα περιλαμβάνει κώδικα, εκτός αν είναι code snippet μέχρι 10 γραμμές και ακολουθείται από τον αντίστοιχο σχολιασμό. Η αναφορά πρέπει να περιλαμβάνει block diagrams που να αποτυπώνουν καλά και περιεκτικά τη δουλειά σας, π.χ. κάποια σήματα και συνδέσεις που θεωρείτε σημαντικά να δείξετε, ή, νέα modules που φτιάξατε που δεν αναφέρονται ρητά στην εκφώνηση. Επίσης, καταγράψτε **αν κάποια κυκλώματα ή μέρη κυκλωμάτων δεν τα υλοποιήσατε, γιατί, και τι προβλήματα συναντήσατε, σε μια παράγραφο που θα της δώσετε χαρακτηριστικό όνομα π.χ. “Προβλήματα”**.

2. Κώδικας VHDL

Εντός του φακέλου SOURCES φτιάξτε 11 φακέλους με τα παρακάτω ονόματα (όλοι με κεφαλαία γράμματα και ακριβώς 11 φάκελοι - μην αλλάξετε τα ονόματα). Όλα τα αρχεία .vhd που έχετε δημιουργήσει, μαζί με τα testbenches και τα libraries που δημιουργήσατε, θα τα βάλετε μέσα σε αυτούς τους φακέλους. Η υποβολή των testbenches είναι υποχρεωτική και χωρίς αυτά η υποβολή θεωρείται ελλιπής. Τα testbenches θα πρέπει να ελέγχουν όλες τις λειτουργίες του component για το οποίο έχουν φτιαχτεί, πχ. εγγραφή και ανάγνωση όλων των καταχωρητών του Register File. Όπου ο πλήρης έλεγχος δεν είναι εφικτός π.χ. στην ALU, δείξτε τα corner cases π.χ. overflow, carry, zero flag.

Περιεχόμενα φακέλου SOURCES

1. Φάκελος **REG**: αρχείο/α .vhd για τον καταχωρητή + testbench
2. Φάκελος **RF**: αρχείο/α .vhd για το αρχείο καταχωρητών + testbench
3. Φάκελος **ALU**: αρχείο/α .vhd για την ALU + testbench
4. Φάκελος **MEM**: αρχείο/α .vhd για το module μνήμης + testbench
5. Φάκελος **IFSTAGE**: IFSTAGE.vhd **κ.α.** + testbench (IFSTAGE_TB.vhd)
6. Φάκελος **DECSTAGE**: DECSTAGE.vhd **κ.α.** + testbench (DECSTAGE_TB.vhd)
7. Φάκελος **EXSTAGE**: EXSTAGE.vhd **κ.α.** + testbench (EXSTAGE_TB.vhd)
8. Φάκελος **MEMSTAGE**: MEMSTAGE.vhd **κ.α.** + testbench (MEMSTAGE_TB.vhd)
9. Φάκελος **DATAPATH**: DATAPATH.vhd **κ.α.** + testbench (DATAPATH_TB.vhd)
10. Φάκελος **CONTROL**: CONTROL.vhd **κ.α.** + testbench (CONTROL_TB.vhd)
11. Φάκελος **PROC_SC**: PROC_SC.vhd **κ.α.** + testbench (PROC_SC_TB.vhd)

- Το **κ.α.** σημαίνει πως ο φάκελος πρέπει και περιέχει **και ότι άλλα αρχεία** χρειάζονται, όπως είναι τα components που κάνετε instantiate. Σε κάθε φάκελο βάζετε μόνο αρχεία με προέκταση .vhd (το .vhd με μικρά γράμματα). Αν φτιάξατε και κάποια δική σας library, να την συμπεριλάβετε και αυτή στον αντίστοιχο φάκελο **που έχετε βάλει τα αρχεία .vhd.**
- Κανένας από τους 11 παραπάνω φακέλους δεν πρέπει να περιέχει υποφακέλους.
- Κάθε φάκελος από τους παραπάνω θα περιέχει ένα (1) μόνο testbench : το τελικό με το οποίο ελέγξατε το αντίστοιχο component, π.χ. τον REG, τον RF, την ALU κλπ.
- Αν έχετε επιπλέον αρχεία .vhd, testbenches, αρχεία αρχικοποίησης της μνήμης, ή δημιουργήσατε κάποια software προγράμματα που σας βοήθησαν στη δουλειά σας, που πιστεύετε πως είναι καλό να δείξετε, αυτά αποθηκεύστε τα σε έναν 12ο φάκελο μέσα στον SOURCES με όνομα TESTING (κεφαλαία). Για τα αρχεία αυτά χρησιμοποιήστε εύστοχα ονόματα από τα οποία να είναι εύκολα αντιληπτό για τι ακριβώς πρόκειται. Ο φάκελος TESTING μπορεί να περιέχει υποφακέλους, αλλά μην το παρακάνετε! Αν θέλετε βάλτε και ένα αρχείο .txt το οποίο θα περιγράφει συνοπτικά τα περιεχόμενα του φακέλου TESTING.
- Όπου υπάρχουν submodules, αυτά θα περιλαμβάνονται ξανά στους φακέλους όπου χρειάζονται (το αντίστοιχο .vhd αρχείο, χωρίς το testbench του). Για παράδειγμα, στο IFSTAGE.vhd γίνεται port map ενός register. Αυτό σημαίνει πως ο φάκελος IFSTAGE θα περιλαμβάνει και το register.vhd αρχείο. Προφανώς, ο φάκελος PROC_SC θα περιλαμβάνει όλα τα .vhd αρχεία που έχετε δημιουργήσει.
- Βάζετε μόνο τις **τελικές εκδόσεις** των modules σας, π.χ. αν τροποποιήσατε τη διεπαφή σε ένα module για να το συνδέσετε κάπου στο DATAPATH.vhd, τότε στον αντίστοιχο φάκελο DATAPATH βάλτε μόνο το τροποποιημένο module.
- Όσο αφορά κώδικες, μόνο ότι υπάρχει στους παραπάνω φακέλους θα βαθμολογηθεί.
- Επαναλαμβάνουμε ότι το αρχείο που θα υποβάλετε πρέπει να είναι της μορφής .zip, π.χ. "a_christodoulou_2019123456.zip". *Όχι* .rar, *ούτε* 7.z ή οτιδήποτε άλλο. Αν έχετε υποβάλλει λάθος, να ξαναυποβάλλετε πριν την προθεσμία.
- Με την αποσυμπίεση του αρχείου θα προκύπτει ένας μόνο φάκελος π.χ. a_christodoulou_2019123456. Ο φάκελος αυτός θα περιέχει 3 φακέλους μόνο: REPORT, SOURCES, WAVEFORMS.
- Αν έχετε πολλά rom.data, βάλτε τα στον φάκελο TESTING (θα τον έχετε δημιουργήσει μέσα στον SOURCES). Και στον WAVEFORMS μπορείτε να τα βάλετε. Π.χ. rom1.data, rom2.data, rom3.data κλπ.
- Ένα (1) αρχείο rom.data μπορείτε να το βάλετε μέσα στους υποφακέλους με τα αρχεία .vhd όπου χρησιμοποιείτε το module της μνήμης και χρειάζεται αρχικοποίηση.

3. Κυματομορφές προσομοίωσης και προγράμματα αναφοράς

Στον φάκελο WAVEFORMS βάζετε τις κυματομορφές προσομοίωσης. Θα σας βοηθήσει η χρήση του Waveform Configuration File (.wcfg) που προσφέρει η Xilinx, για ν' αποθηκεύετε τα σήματα σε αρχείο. Στον ίδιο φάκελο θα βρίσκονται και τα προγράμματα αναφοράς που δημιουργήσατε, συνοδευόμενο το καθένα από ένα text αρχείο που θα περιλαμβάνει τις assembly εντολές του προγράμματος αναφοράς. Στον φάκελο WAVEFORMS δεν θα

υπάρχουν υποφάκελοι. Σημείωση: το αρχείο .wcfg το κάνετε “Load” μετά από compile, για να εμφανίζονται αυτόματα τα σήματα που θέλετε να παρακολουθείτε στην προσομοίωση.