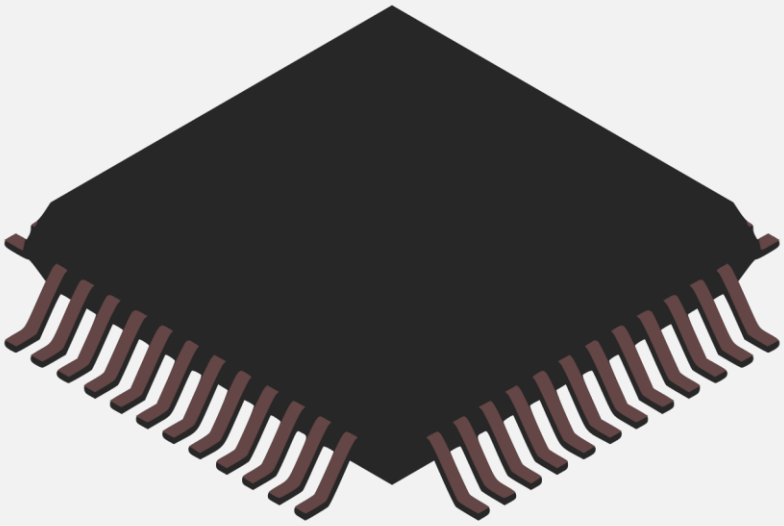
ΟΡΓΑΝΩΣΗ ΥΠΟΛΟΓΙΣΤΩΝ

Αναφορά 5ης Εργαστηριακής Άσκησης

«ΜΕΤΑΤΡΟΠΗ ΤΟΥ CHARIS-4 ΣΕ PIPELINE PROCESSOR»



Ζαχαριουδάκης Νικόλας 2016030073
Γαλάνης Μιχάλης 2016030036

ΠΕΡΙΕΧΟΜΕΝΑ

Οι σύνδεσμοι για τις παρακάτω ενότητες είναι διαδραστικοί.
Πιέστε πάνω στο επιθυμητό τμήμα για τη μετάβαση σε αυτό.

ΕΙΣΑΓΩΓΗ

Σκοπός Εργαστηρίου

Σκοπός της 5ης και τελευταίας άσκησης είναι η μετατροπή του CHARIS-4 σε 5-stage pipeline processor για την μεγιστοποίηση του throughput του επεξεργαστή μας. Οι απαραίτητες βάσεις για να μπορεί να υλοποιηθεί η διαδικασία αυτή έχουν γίνει από προηγούμενες ασκήσεις (π.χ ολοκλήρωση Control & Datapath). Χρειάστηκαν όμως αρκετές αλλαγές τόσο στο Control όσο και στο Datapath.

Προαπαιτούμενα

Απαιτείται η άπταιστη γνώση της VHDL και εξοικείωση στα εργαλεία σχεδιασμού της Xilinx. Επίσης θα χρησιμοποιηθεί πλήρως το υλικό που παράχθηκε στις προηγούμενες εργαστηριακές ασκήσεις.

Α – ΑΛΛΑΓΕΣ ΣΤΟ DATAPATH

Σχεδίαση & Υλοποίηση

Με την εφαρμογή του pipeline, κάθε κύκλος ρολογιού γίνεται ένα pipe stage. Δηλαδή, ενώ κάθε εντολή χρειάζεται 5 κύκλους ρολογιού για να ολοκληρωθεί, ο επεξεργαστής σε κάθε κύκλο αναλαμβάνει την εκτέλεση συγκεκριμένων τμημάτων διαφορετικών εντολών. Η διαδικασία φαίνεται συνοπτικά στον επόμενο πίνακα.

Εντολή	Κύκλος Ρολογιού								
	1	2	3	4	5	6	7	8	9
i	**IF**	**ID**	**EX**	**MEM**	**WB**				
i + 1		**IF**	**ID**	**EX**	**MEM**	**WB**			
i + 2			**IF**	**ID**	**EX**	**MEM**	**WB**		
i + 3				**IF**	**ID**	**EX**	**MEM**	**WB**	
i + 4					**IF**	**ID**	**EX**	**MEM**	**WB**

Για να γίνει το Datapath μας pipelined, απαιτούνται διάφορες τιμές να περνάνε από ένα στάδιο του pipeline στο επόμενο. Για αυτό το λόγο, δημιουργούμε καταχωρητές (pipeline registers). Πρέπει επίσης να προσέξουμε τα εξής πράγματα:

- Το Register File χρησιμοποιείται σε δύο διαφορετικά στάδια, στο **ID** και στο **WB**. Οπότε χρειάζεται να εφαρμόσουμε δύο αναγνώσεις και μία εγγραφή σε κάθε κύκλο ρολογιού. Πρόβλημα που μπορεί να προκύψει -> **Data Hazard** (Η λύση παρουσιάζεται αργότερα).
- Για να ξεκινήσουμε μια εντολή, πρέπει να αυξήσουμε και να αποθηκεύσουμε το PC σε κάθε κύκλο ρολογιού κατά τη διάρκεια του **IF** stage.
- Το Datapath του CHARIS-4 χρησιμοποιεί μια κοινή μνήμη για εντολές και δεδομένα. Μπορεί δηλαδή να χρησιμοποιηθεί σε δύο διαφορετικά στάδια, στο **IF** και στο **MEM** όταν έχουμε

εντολές ανάγνωσης/εγγραφής μνήμης. Πρόβλημα που μπορεί να προκύψει λοιπόν -> **Structural Hazard** (Η λύση παρουσιάζεται αργότερα).

Για τα 5 στάδια του pipeline μας, δημιουργούμε δηλαδή 4 pipeline registers (που ενώνουν αυτά τα 5 στάδια μεταξύ τους) τα οποία παρουσιάζονται συνοπτικά παρακάτω:

- **IF_ID_pipeline:** ενώνει τα στάδια instruction fetch και instruction decode και είναι υπεύθυνο για τη διατήρηση των τιμών του **instruction** (MEM[PC]) και του **PC + 4** (έτσι ώστε στον επόμενο κύκλο ρολογιού να γίνει ανάγνωση της επόμενης εντολής στο FETCH stage).

Τα επόμενα τρία pipelines, εκτός από το να διαχειρίζουν τις τιμές του datapath, διαθέτουν και control signals καθώς αυτά που θα αποθηκευτούν εξαρτώνται από την κάθε εντολή (αφού πλέον την έχουμε αποκωδικοποιήσει και γνωρίζουμε το είδος της).

- **ID_EX_pipeline:** ενώνει τα στάδια του instruction decode και execute. Στο κομμάτι του datapath του αποθηκεύει σε καταχωρητές τις τιμές του **PC+4**, **RF_A**, **RF_B**, **immed**, **RD**, **RS** και **RT**. Όσον αφορά το control απαιτείται η διατήρηση των τιμών **ALU_CTR**, **ALU_SRC**, **EX_MEM_SIG** και **MEM_WB_SIG**.
- **EX_MEM_pipeline:** ενώνει τα στάδια του execution με το memory write. Τα control signals που διατηρούνται είναι τα **EX_MEM_SIG** και **MEM_WB_SIG**. Στο Datapath οι τιμές που διατηρούνται είναι τα **PC+4**, **Zero** (1-bit), **ALU_out**, **RF_B** και **RD**.
- **MEM_WB_pipeline:** ενώνει τα στάδια του memory με το write back. Διατηρείται το **EX_MEM_SIG** για control και τα **ALU_out**, **RF_B** και **RD** για datapath.

Μεχρι τώρα, η υλοποίηση προσφέρει μεγαλύτερο throughput λόγω παραλληλισμού σε χαμηλό επίπεδο (επίπεδο εντολών). Ο παραλληλισμός αυτός όμως φέρνει και «παρενέργειες» σε ένα μη-ιδανικό pipeline. Υπάρχουν περιπτώσεις όπου παρεμποδίζεται η ολοκλήρωση κάποιας εντολής όπως στα παραδείγματα που αναφέρθηκαν προηγουμένως. Σε αυτές τις περιπτώσεις λέμε ότι έχουμε pipeline hazards.

Από τα τρία είδη που υπάρχουν (Structural, Data, Control) εμάς θα μας απασχολήσουν τα πρώτα δύο καθώς τα Control Hazards παρουσιάζονται σε εντολές που περιέχουν διακλαδώσεις. Ως αντιμετώπιση σε αυτά τα Hazards, εφαρμόζουμε τα λεγόμενα **stalls** ή αλλιώς καθυστερήσεις στη κανονική ροή του pipeline, οι οποίες έχουν προφανώς ένα κόστος στην ταχύτητα του. Μόνο για τη περίπτωση των Data Hazards και για τη περίπτωση που έχουμε εντολές τύπου ADD / SUB, μπορούμε να αποφύγουμε stalls με τη τεχνική **forwarding** που θα εξηγηθεί αργότερα. Στον επόμενο πίνακα φαίνεται πως επηρεάζονται τα στάδια του pipeline στη περίπτωση που εφαρμόσουμε stall (πχ. Structural Hazard όπου i είναι εντολή load/store).

Εντολή	**Κύκλος Ρολογιού**									
	1	2	3	4	5	6	7	8	9	10
i	**IF**	**ID**	**EX**	**MEM**	**WB**					
i + 1		**IF**	**ID**	**EX**	**MEM**	**WB**				
i + 2			**IF**	**ID**	**EX**	**MEM**	**WB**			
i + 3				**STALL**	**IF**	**ID**	**EX**	**MEM**	**WB**	
i + 4						**IF**	**ID**	**EX**	**MEM**	**WB**

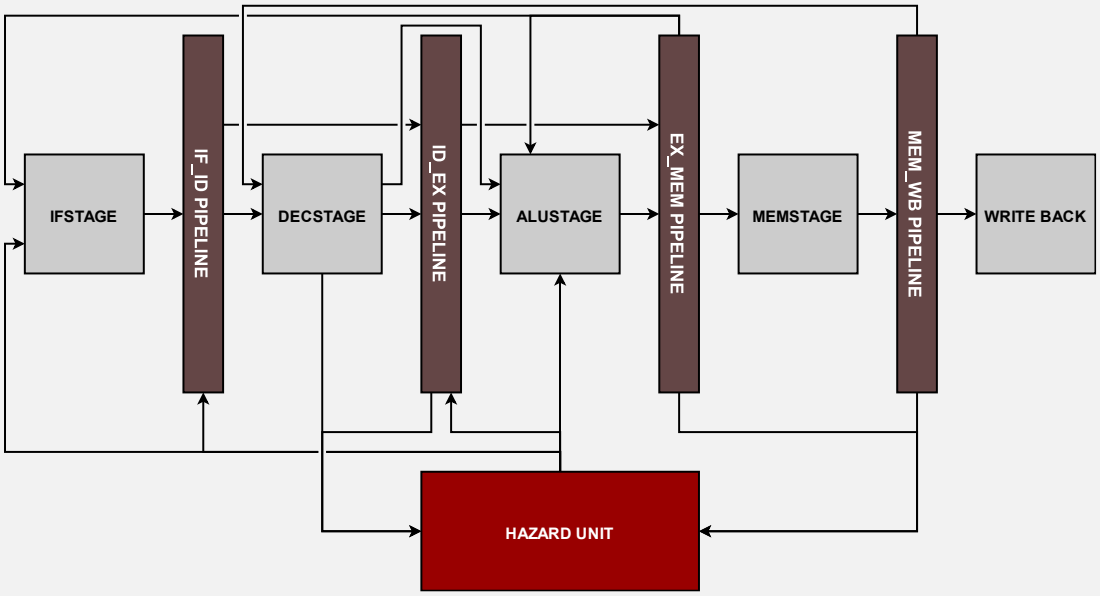
Σύμφωνα με τον πίνακα αυτόν, όταν έχουμε συνεχόμενες εντολές ADD/SUB που χρησιμοποιούν τους ίδιους καταχωρητές έχουμε Data Hazard διότι η εντολή i αποθηκεύει στον RF το αποτέλεσμα στον κύκλο 5 (WB stage) ενώ η εντολή i+1 διαβάζει το αποτέλεσμα στον κύκλο 2 (ID stage) πριν αποθηκευτεί η σωστή τιμή. Παρατηρούμε όμως ότι το αποτέλεσμα στην εντολή i παράγεται ήδη από τον κύκλο 3, και στη πραγματικότητα η εντολή i+1 χρειάζεται το αποτέλεσμα αυτό στον κύκλο 4. Συνεπώς μπορούμε χωρίς να καθυστερήσουμε τη ροή του pipeline, να κάνουμε «forward» το αποτέλεσμα της ALU της εντολής i ως είσοδο στην ALU της επόμενης εντολής.

Τα stalls και τα forwards στην υλοποίησή μας τα διαχειριζόμαστε από το module Hazard_Unit που βρίσκεται εντός του Datapath. Ο κώδικας που τον υλοποιεί παρουσιάζεται παρακάτω:

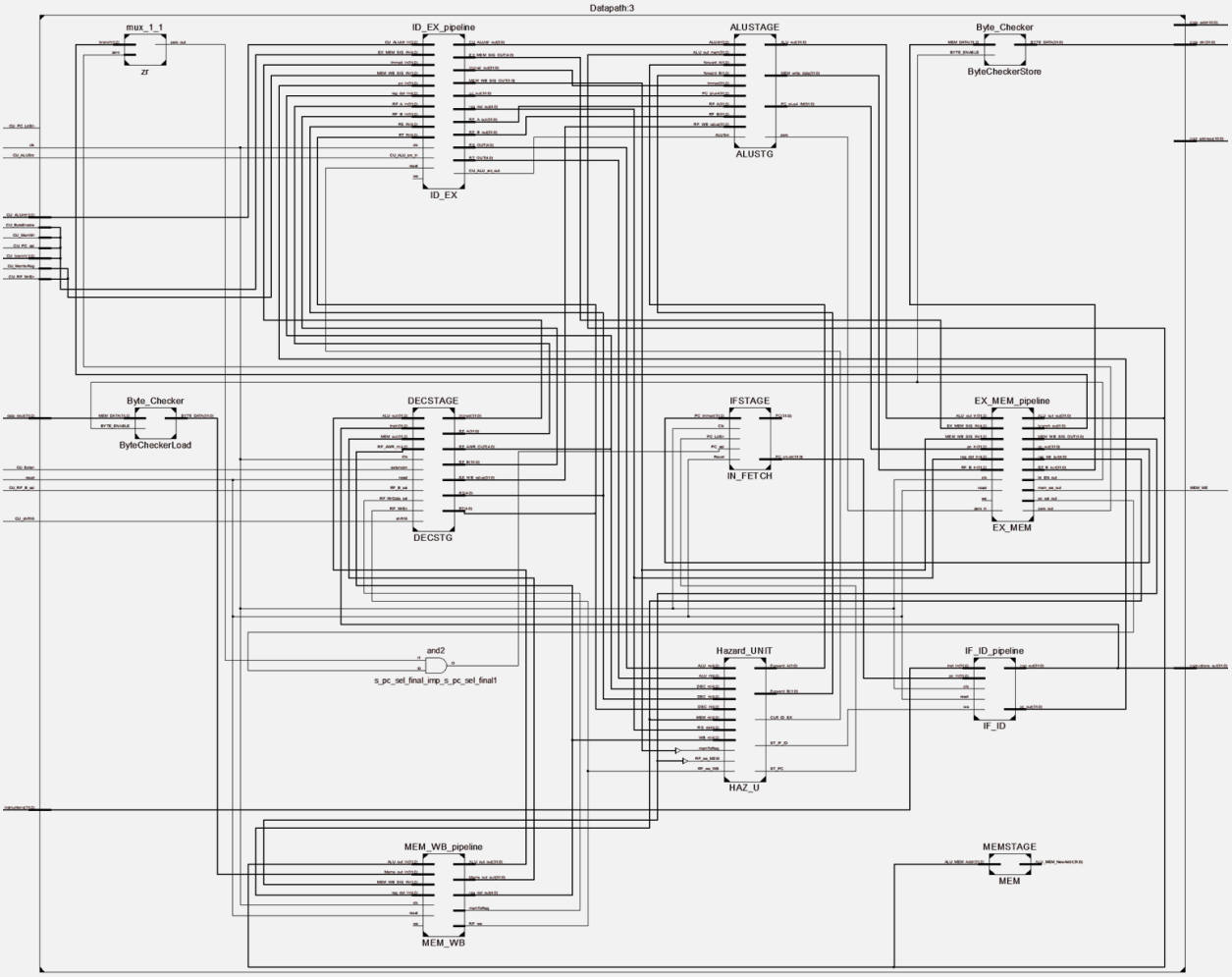
```
begin
	forwarding: process(RF_we_MEM,RF_we_WB,ALU_rs,ALU_rt,MEM_rd,WB_rd)
	begin
		if ((RF_we_MEM='1') and (MEM_rd /= "00000") and (MEM_rd = ALU_rs)) then
			Forward_A <= "10";
		elsif ((RF_we_WB='1') and  (WB_rd /= "00000") and (WB_rd = ALU_rs)
					and not  ((RF_we_MEM='1') and (MEM_rd /= "00000")and
(MEM_rd = ALU_rs))) then
			Forward_A <= "01";
		else
			Forward_A <= "00";
		end if;
		if (((RF_we_MEM='1') and (MEM_rd /= "00000") and ((MEM_rd = ALU_rt) or
(MEM_rd = RG_dst))))then
			Forward_B <= "10";
		elsif ((((RF_we_WB='1') and  (WB_rd /= "00000") and ((WB_rd = ALU_rt) or
(WB_rd = RG_dst)))
					and not ((RF_we_MEM='1') and (MEM_rd /= "00000") and
((MEM_rd = ALU_rt) or (MEM_rd = RG_dst))))) then
			Forward_B <= "01";
		else
			Forward_B <= "00";
		end if;

	end process;
	stalling: process(memToReg)
	begin
	if (memToReg='1' and ((DEC_rs=RG_dst) or (DEC_rt=RG_dst) or (DEC_rd=RG_dst)))
then
		ST_PC  <= '0';
		ST_IF_ID <= '0';
		CLR_ID_EX<= '1';
	else
		ST_PC  <= '1';
		ST_IF_ID <= '1';
		CLR_ID_EX<= '0';
	end if;
	end process;
end Behavioral;
```

Στην επόμενη σελίδα παρουσιάζουμε δύο block diagrams που περιγράφουν όσα έχουμε μελετήσει παραπάνω. Το πρώτο είναι υψηλού επιπέδου και δείχνει χωρις μεγάλη λεπτομέρεια τη διαδικασία και την επικοινωνία μεταξύ των βαθμίδων, των pipelines και του Hazard Unit ενώ στο δεύτερο φαίνεται σε χαμηλότερο επίπεδο ολόκληρο το Datapath με όλες τις καλωδιώσεις.

HIGH LEVEL BLOCK DIAGRAM



LOW LEVEL BLOCK DIAGRAM

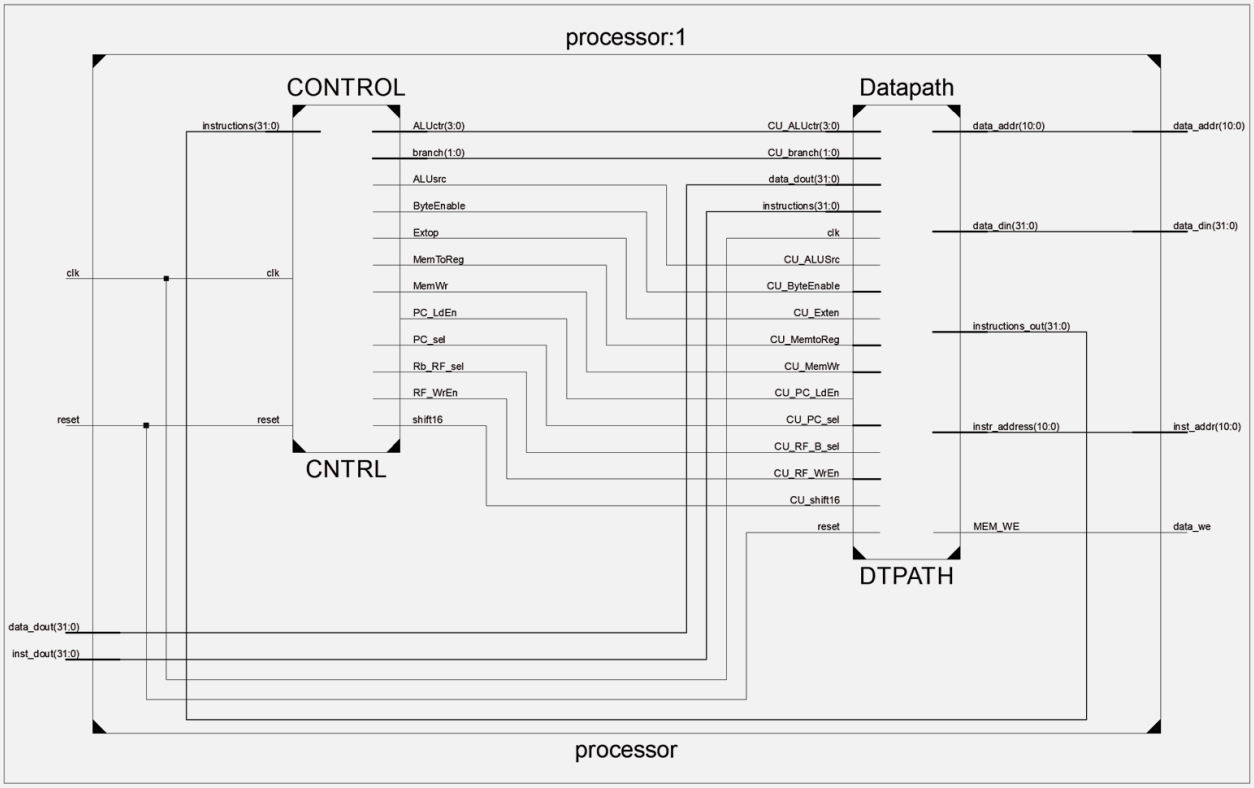


Β – ΑΛΛΑΓΕΣ ΣΤΟ CONTROL

Σχεδίαση & Υλοποίηση

Όσον αφορά το Control Unit, καταργήσαμε αρχικά την FSM καθώς πλέον η λογική διαχειρίζεται στο pipelined datapath και συγκεκριμένα στα 4 pipelines που δημιουργήσαμε. Επιστρέψαμε στην παλιά εκδοχή του Control δηλαδή όπου η κάθε έξοδος αποτελούσε ένα process αντί να υπάρχει ένα process για όλες τις εξόδους. Προσθέσαμε επίσης μια 2-bit έξοδο branch.

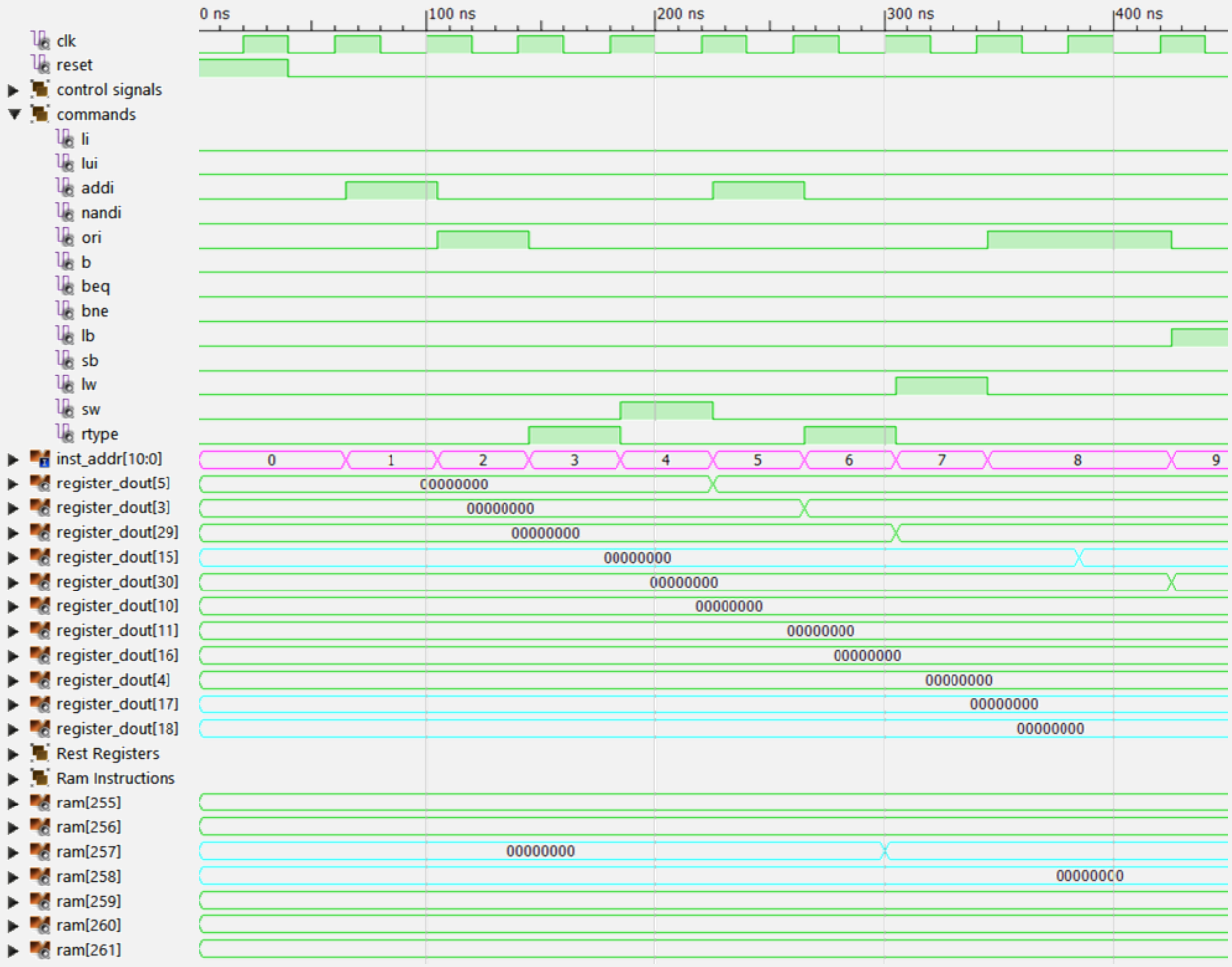
Παρουσιάζεται παρακάτω το Block diagram του επεξεργαστή με το ανανεωμένο Control & Datapath.



Γ – ΕΠΑΛΗΘΕΥΣΗ ΣΥΣΤΗΜΑΤΟΣ

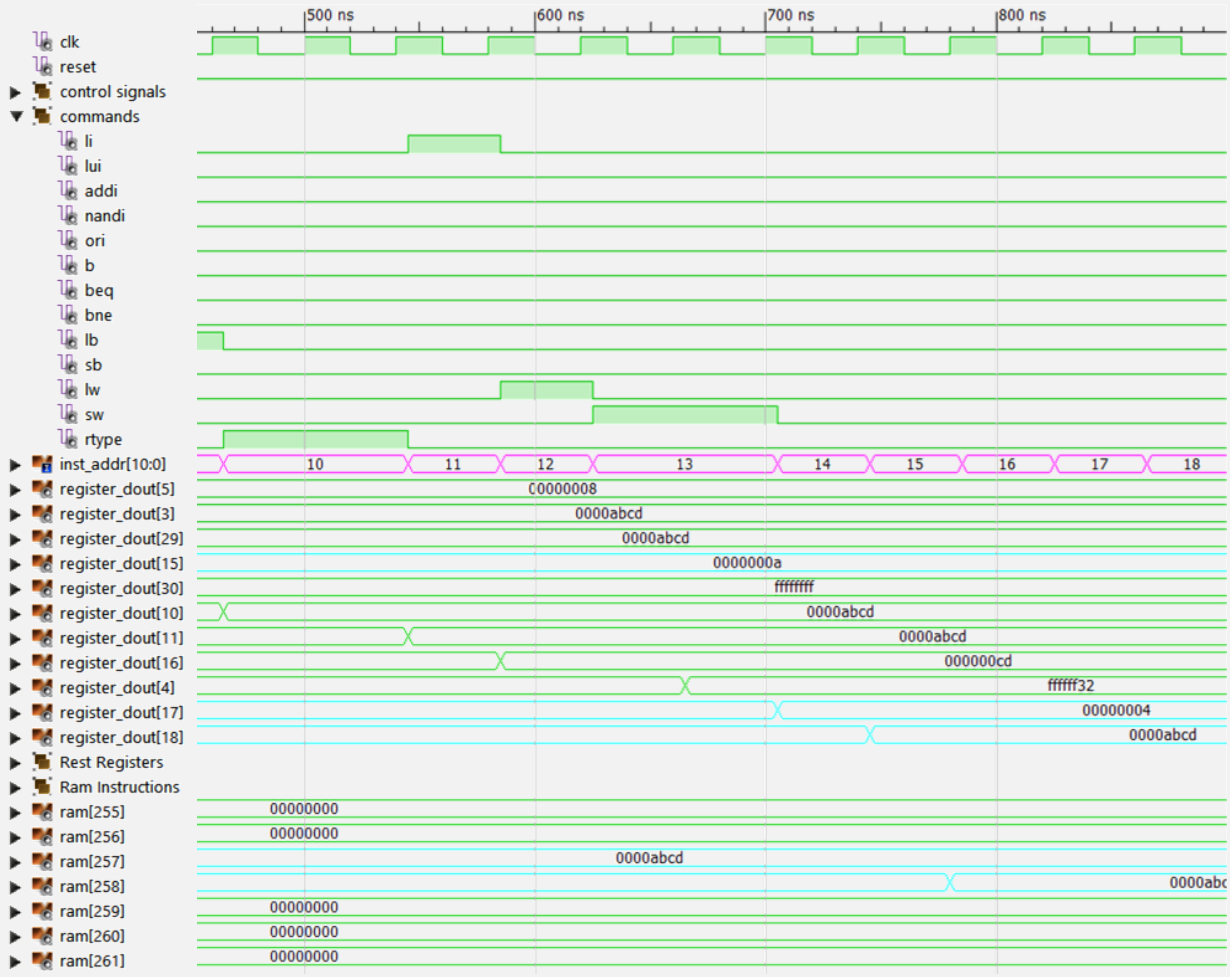
Προσομοίωση ram0.data (Κυματομορφές)

0 ns – 450 ns:



Αρχικά παρατηρούμε ότι γύρω στα 300 ns, ολοκληρώνεται η εντολή **`[add $r29,$r3,$r0]`** έχει αντιμετωπιστεί επιτυχώς το Data Hazard μέσω τεχνικής forwarding καθώς η προηγούμενη εντολή **`[ori $r3,$r0,ABCD]`** δεν έχει προλάβει να ολοκληρωθεί πλήρως έτσι ώστε να έχουμε την τιμή στο register R3. 2η περίπτωση Data Hazard συναντάμε στην 6η και 7η εντολή (zero-based), που είναι οι **`[lw $r10,-5($r5)]`** και **`[ori $r11,$r10,0]`** αντίστοιχα, όπου θέλουμε να χρησιμοποιήσουμε το περιεχόμενο του καταχωρητή R10. Το forwarding δεν ευνοεί σε αυτή την περίπτωση αφού η εντολή lw έχει latency 2 κύκλων (δηλαδή απαιτούνται 2 κύκλοι έτσι ώστε να μπορεί να χρησιμοποιηθεί το περιεχόμενό της. Χρησιμοποιούμε λοιπόν τεχνική stalling και θέτουμε σε παύση όλες τις επόμενες εντολές για το διάστημα αυτό

450 ns – 900 ns:



Στις τελευταίες 3 εντολές (**`[li $r17,4],[lw $r18,($r17)],[sw $r18,4($r17)]`**) μπορεί εύκολα να καταλάβει κανείς ότι εφαρμόζουμε forwarding από την li στην lw και sw λόγω του περιεχομένου του R17. Παράλληλα, έχουμε stalling λόγω της εντολής lw για το R18 τον οποίο θέλουμε να αποθηκεύσουμε στη θέση 4(R17).