ΠΡΟΗΓΜΕΝΗ ΣΧΕΔΙΑΣΗ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ 2012 – 2013

ΚΑΘ. ΑΝΤΩΝΗΣ ΠΑΣΧΑΛΗΣ

ΥΛΟΠΟΙΗΣΗ RISC ΕΠΕΞΕΡΓΑΣΤΗ ΑΡΧΙΤΕΚΤΟΝΙΚΗΣ MIPS R2000 ΜΕ ΕΝΣΩΜΑΤΩΜΕΝΕΣ ΜΕΘΟΔΟΥΣ ΔΟΚΙΜΗΣ ΣΤΟ ΥΛΙΚΟ ΚΑΙ ΤΟ ΛΟΓΙΣΜΙΚΟ (BIST / SBST)

ΕΠΩΝΥΜΟ: Κουσκουμβεκάκης
ΟΝΟΜΑ: Ηλίας
Μ.Μ.: 210

ΕΠΩΝΥΜΟ: Στεργίου
ΟΝΟΜΑ: Παναγιώτης
Μ.Μ.: 200

ΙΟΥΛΙΟΣ 2013

Περιεχόμενα

1. Εισαγωγή

1.1 Διατύπωση του προβλήματος

Στην παρούσα εργασία υλοποιήσαμε έναν επεξεργαστή MIPS των 32 bit ο οποίος υποστηρίζει το σύνολο εντολών του ιστορικού επεξεργαστή MIPS R2000. Κάθε εντολή εκτελείται σε 3 ή περισσότερους κύκλους ρολογιού χωρίς να υποστηρίζεται η διοχέτευση (pipeline) των εντολών μεταξύ των διαφορετικών σταδίων. Όλες οι λειτουργίες του επεξεργαστή ενορχηστρώνονται κατά τη διάρκεια των κύκλων εκτέλεσης της κάθε εντολής από τη μονάδα ελέγχου η οποία παράγει τα κατάλληλα σήματα συγχρονισμού των μονάδων που απαρτίζουν τον επεξεργαστή.

Οι εντολές φορτώνονται από μια μνήμη εντολών μεγέθους 64KBits η οποία μπορεί να φορτωθεί με 2K εντολές. Για την επεξεργασία και αποθήκευση των δεδομένων χρησιμοποιείται ένα αρχείο 32 καταχωρητών, γενικής χρήσης, των 32 bits και μια μνήμη εντολών χωρητικότητας 2K λέξεων των 32 bits (συνολικά 64KBits).

Η μονάδα εκτέλεσης αριθμητικών και λογικών πράξεων (ALU) υποστηρίζει όλες τις γνωστές πράξεις μεταξύ ακεραίων αριθμών συμπεριλαμβανομένης και της πράξης του πολλαπλασιασμού χρησιμοποιώντας είτε διοχετευμένο πολλαπλασιαστή είτε κανονικό. Η επιλογή της μορφής του πολλαπλασιαστή γίνεται εύκολα από μια generic παράμετρο. Η μονάδα αυτή έχει επίσης τη δυνατότητα να εκτελέσει έλεγχο ορθής λειτουργίας της μονάδας του πολλαπλασιαστή χρησιμοποιώντας 3 τεχνικές ενσωματωμένης αυτοδοκιμής (BIST). Οι τεχνικές αυτές είναι οι LFSR, Deterministic (Counter) και ATPG.

Για την σχεδίαση και υλοποίηση του επεξεργαστή χρησιμοποιήθηκαν τα εργαλεία της Xilinx (ISE 14.5) καθώς και ο εξαιρετικός VHDL editor Sigasi. Για την επαλήθευση της ορθής λειτουργίας του επεξεργαστή χρησιμοποιήσαμε τον ενσωματωμένο στο ISE simulator (Isim). Τέλος υλοποιήσαμε μερικά προγράμματα σε assembly τα οποία φορτώσαμε και εκτελέσαμε στον επεξεργαστή με επιτυχία. Για την αρχικοποίηση της μνήμης δεδομένων δημιουργήσαμε software σε C το οποίο καλείται από δικό μας Makefile και αναλαμβάνει το γέμισμα της μνήμης εντολών με machine code. Η μετατροπή της assembly σε machine code έγινε χρησιμοποιώντας τα GNU Binutils (as, ld) μεταγλωτισμένα για την αρχιτεκτονική MIPS (cross-compiled).

Σύνολο εντολών που υλοποιούνται

Εντολή	Αριθμός κύκλων	Περιγραφή
LW	5	Load Word
LH	5	Load Half Word
LHU	5	Load Half Word Unsigned
LB	5	Load Byte
LBU	5	Load Byte Unsigned
SW	4	Store Word
SH	4	Store Half Word
SB	4	Store Byte
ADDI	4	Add Immediate
ADDIU	4	Add Immediate Unsigned
ANDI	4	And Immediate

ORI	4	Or Immediate
XORI	4	Xor Immediate
ADD	4	Add
ADDU	4	Add Unsigned
SUB	4	Subtract
SUBU	4	Subtract Unsigned
AND	4	And
OR	4	Or
NOR	4	Nor
XOR	4	Xor
MULT	8 (pipelined) or 5 (normal)	Multiply
MFHI	2	Move from Hi
MFLO	2	Move from Lo
MTHI	3	Move to Hi
MTLO	3	Move to Lo
SLL	4	Shift Left Logical
SRL	4	Shift Right Logical
SRA	4	Shift Right Arithmetic
SLLV	4	Shift Left Logical Variable
SRLV	4	Shift Right Logical Variable
SRAV	4	Shift Right Arithmetic Variable
LUI	4	Load Upper Immediate
SLTI	4	Set Less Than Immediate
SLTIU	4	Set Less Than Immediate Unsigned
SLT	4	Set Less Than
SLTU	4	Set Less Than Unsigned
BEQ	4	Branch on Equal
BNE	4	Branch on Not Equal
BLEZ	4	Branch on Less than Equal Zero
BGTZ	4	Branch on Greater than Zero
BLTZ	4	Branch on Less than Zero
BGEZ	4	Branch on Greater than Equal Zero
JR	3	Jump Regiter
JALR	3	Jump And Link Register
J	3	Jump
JAL	3	Jump And Link
TEST	-	Test Multiplier with BIST methods

1.2 Μεθοδολογία

Κατά τη σχεδίαση και υλοποίηση του έργου προσπαθήσαμε να εφαρμόσουμε τις ακόλουθες τρείς αρχές: Απλότητα, Ορθότητα και Βελτιστοποίηση στο τέλος. Η κάθε μια από αυτές τις αρχές θεωρούμε οτι είναι εξίσου σημαντική και η σωστή εφαρμογή της καθεμιάς βοηθάει στην επίτευξη της προηγούμενης.

Για παράδειγμα η συγγραφή απλού και καθαρού κώδικα VHDL βοηθάει πάρα πολύ όλους όσους εμπλέκονται με το project και μας επιτρέπει να κατανοούμε σε βάθος τη λειτουργία του χωρίς την ανάγκη για συνεχείς διευκρινήσεις μεταξύ μας. Επίσης οποιεσδήποτε μελλοντικές αλλαγές χρειάζονται είναι πολύ πιθανό να γίνουν εύκολα και χωρίς πολύ κόπο.
Η ορθότητα του σχεδίου είναι άμεσα συνεδεμένη με την απλότητα του. Ως γνωστόν είναι πολύ εύκολο κάποιος να γράψει κώδικα VHDL ο οποίος θα παράγει διαφορετικά αποτελέσματα στη σύνθεση από την προσομοίωση. Η συγγραφή απλού και με συγκεκριμένο τρόπο κώδικα βοηθάει τον Synthesizer να παράξει σωστό και χωρίς προβλήματα κύκλωμα οι αποκρίσεις και τα αποτελέσματα του οποίου θα συμβαδίζουν άμεσα με αυτό που προσομοιώνουμε. Συνέπεια αυτού είναι οτι στο τέλος τόσο η functional όσο και η timing (par) προσομοιώσεις έχουν μεγάλη πιθανότητα να συμβαδίζουν ώστε να μη χρειαστεί κάποια παρέμβαση στον κώδικα για να επαληθευτεί η λειτουργία του σχεδίου και να εξασφαλιστεί οτι θα λειτουργήσει ορθά μέσα στο FPGA. Αυτό ακριβώς συνέβει και στην περίπτωση μας όπου για να λειτουργήσει ορθά η timing προσομοίωση χρειάστηκε να κάνουμε αλλαγή σε μόλις μια γραμμή κώδικα από όλο το σχέδιο των χιλιάδων γραμμών. Αυτή η λάθος γραμμή δε μας πήρε πάνω από 1 ώρα για να την ανιχνέυσουμε και να την διορθώσουμε.

Με δεδομένο οτι έχουμε ένα πλήρως ορθό και λειτουργικό σχέδιο το οποίο περνάει με επιτυχία τις timing (par) προσομοιώσεις, μπορούμε να ασχοληθούμε με τη βελτιστοποίηση του παρατηρώντας με ιδιαίτερη προσοχή τα αποτελέσματα των καθυστερήσεων των μονάδων στο γνωστό static timing report που παράγουν τα εργαλεία σύνθεσης και υλοποίησης του σχεδίου. Μερικές από τις αλλαγές βελτιστοποίησης που κάναμε είναι η τοποθέτηση registers σε επιλεγμένες λειτουργικές μονάδες. Οι πιο σημαντικές από αυτές ήταν η μονάδα control combinational και η μονάδα του πολλαπλασιαστή όπου επιλέχθηκε η χρήση ενός pipelined πολλαπλασιαστή.

Αφού σιγουρευτήκαμε οτι ο επεξεργαστής μπορεί και εκτελεί σωστά και με τον ταχύτερο δυνατό τρόπο όλες τις υποστηριζόμενες εντολές, η προσοχή μας στράφηκε στις τεχνικές δοκιμαστικότητας του πολλαπλασιαστή όπως είναι η LFSR, η Deterministic Counter και η ATPG. Όλες οι τεχνικές αυτές υλοποιήθηκαν τόσο στο hardware (BIST) όσο και σε software (SBST). Η δοκιμαστικότητα των επιμέρους μονάδων είναι πολύ σημαντική ιδιαίτερα στην εποχή που ζούμε με τα περισσότερα σχέδια να είναι εξαιρετικά μεγάλου μεγέθους και πολυπλοκότητας. Η υλοποποίηση και η δοκιμή των παραπάνω τεχνικών μας επέτρεψε να εξοικοιωθούμε με την εφαρμογή δοκιμαστικότητας σε μονάδες τους σχεδίου μας και να μπορούμε πλέον να τις εφαρμόσουμε σε οποιοδήποτε μελλοντικό σχέδιο. Αυτό θα μας επιτρέπει να έχουμε γρήγορα και άμεσα μια εικόνα για τον αν η μονάδα στην οποία εφαρμόζονται λειτουργεί ορθα ή λανθασμένα κατά οποιοδήποτε τρόπο ή αιτία ακόμα και τη στιγμή που το σύστημα που περιέχει τη μονάδα είναι σε λειτουργία.

1.3 Καταμερισμός εργασίας

Για την ολοκλήρωση της παρούσας εργασίας έγινε καταμερισμός των απαραίτητων εργασιών ώστε να πετύχουμε ένα ποιοτικό αποτέλεσμα σε εύλογο χρονικό διάστημα. Αν και στην αρχή ο καθένας μας ανέλαβε συγκεκριμένες μονάδες προς υλοποίηση και δοκιμή, στην πορεία ο καταμερισμός άλλαξε δυναμικά, ιδιαίτερα στη φάση των δοκιμών και της επαλήθευσης ορθής λειτουργίας όπου και οι δύο μας βρεθήκαμε να κάνουμε αλλαγές σε κομμάτια που αρχικά δεν είχαμε συμμετοχή. Επίσης προσπαθήσαμε να αναλάβει ο καθένας μας κομμάτια ανάλογα με την προηγούμενη εμπειρία του από αντίστοιχα μαθήματα και εργασίες στο μεταπτυχιακό που ακολουθούμε. Παρακάτω παραθέτουμε έναν ενδεικτικό πίνακα του καταμερισμού των εργασιών μεταξύ μας.

Ηλίας Κ.	**Παναγιώτης Σ.**
Design Implementation an Simulation	
Control FSM	CPU Top
Control Combinational	Instruction and Data Memory
Datapath (Top and Registers)	Control Combinational
Register Multiplexer	Register File
NPC (Selector and Mux)	Extend Immediate
DM Control	NPC (Increment, PSD, SL2, Adder)
ALU Mult (Top, Control, Unit, MISR, Comparator)	ALU Top, Shifter, Multiplexer
	ALU Mult (LFSR, Counter, ATPG)
Software	
full.s, simple.s	matrix.s, lfsr.s, counter.s, atpg.s
Assembly to BRAM Toolchain	

2. Τεχνική Περιγραφή

Κατά την υλοποίηση του ζητούμενου επεξεργαστή χρειάστηκε να δημιουργηθούν πολλές και διαφορετικές μονάδες οι οποίες συνδυαστικά επιτυγχάνουν την εκτέλεση όλων των υποστηριζόμενων εντολών. Μπορούμε να χωρίσουμε όλες αυτές τις μονάδες στις εξής κατηγορίες:

Instruction Memory:

Η μνήμη εντολών είναι υλοποιημένη χρησιμοποιώντας και ενώνοντας σειριακά 4 BRAMs των 16KBit (2K x 8) η καθεμία για συνολικό μέγεθος 64 KBits.

Data Memory:

Η μνήμη δεδομένων είναι υλοποιημένη χρησιμοποιώντας και ενώνοντας παράλληλα 4 BRAMs των 16KBit (2K x 8) η καθεμία για συνολικό μέγεθος 64 KBits.

Control Unit:

Η μονάδα ελέγχου είναι υλοποιημένη σε δύο ξεχωριστά τμήματα, ένα συνδυαστικό και ένα ακολουθιακό. Είναι υπεύθυνη για τη δημιουργία όλων τον σημάτων ελέγχου και συγχρονισμού των υπολοίπων μονάδων.

Datapath:

Περιλαμβάνει όλες τις λειτουργικές μονάδες, ο συνδυασμός και συγχρονισμός των οποιών επιτρέπει την εκτέλεση της κάθε εντολής ξεχωριστά σε 3 η περισσότερους κύκλους. Περιλαμβάνει επίσης και τη μονάδα ALU η οποία είναι υπεύθυνη για την εκτέλεση όλων των αριθμητικών και λογικών πράξεων. Για την περίπτωση της πράξης του πολλαπλασιασμού υποστηρίζεται η ενσωματωμένη αυτοδοκιμή του πολλαπλασιαστή με φόρτωση διανυσμάτων ανάλογα τη μέθοδο (LFSR, Deterministic Counter, ATPG) και ελέγχου των αποτελεσμάτων.

2.1 Τεχνική περιγραφή όλων των entities του επεξεργαστή

Datapath Registers

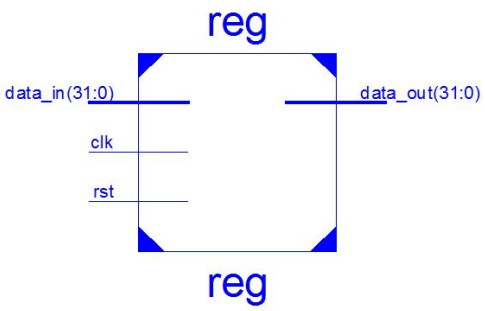
Γενική περιγραφή

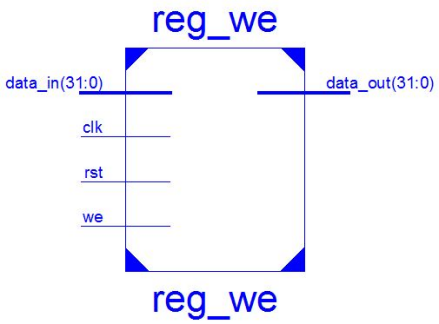
Η δίοδος δεδομένων περιέχει αρκετούς καταχωρητές οι οποίοι είναι υπεύθυνη για την προσωρινή αποθήκευση των δεδομένων μεταξύ των διαφόρων σταδίων εκτέλεσης κάθε εντολής που υλοποιείται. Οι καταχωρητές αυτοί είναι οι εξής:

Όνομα	Μέγεθος (Bits)	Σήμα Εγγραφής	Είσοδος	Έξοδος	Περιγραφή
PC	32	PC Write	NPC Mux	NPC Inc, P	Program Counter
NPC	32	-	NPC Inc	NPC Mux, NPC Adder	Next Program Counter
P	4	-	PC	PSD	PC 4 MSBs
D	32	-	P	NPC Mux	NPC for J, JAL
A	32	-	Register File	ALU Mux, NPC Mux	RF Porta A
B	32	-	Register File	ALU Mux, MDRIN	RF Port B
I	32	-	Extend Immed	ALU Mux, NPC Shift	Immediate extended to 32bits
M	32	-	NPC Adder	NPC Mux	NPC for Branches
S	5	-	IR	ALU	Shift amount
ALUOUT	32	-	ALU	DM Control, RF Mux	ALU result
HI	32	HI Write	ALU	RF Mux	ALU Mult MSB result
LO	32	LO Write	ALU	RF Mux	ALU Mult LSB result
FLAGS	4	-	ALU	NPC Select, Debug Output	Zero, Negative, Overfolow
MDRIN	32	-	B	DM Control	DM word to store
MAR	32	MAR Write	ALU	DM Control	DM Address
ERR	1	-	DM Control	Debug Output	DM Alignment Error

Όλοι οι καταχωρητές αυτοί είναι υλοποιημένοι σε δύο μονάδες (reg, reg_we) με την ίδια περιγραφή συμπεριφοράς ανεξαρτήτως του μεγέθους τους χρησιμοποιώντας μια generic παράμετρου (W) που ορίζει το μέγεθος αυτό. Επίσης αν ο καταχωρητής χρειάζεται κάποιο σήμα ενεργοποίησης εγγραφής (Write Enable) τότε πρέπει να χρησιμοποιηθεί η μονάδα reg_we ενώ αν δε το χρειάζεται χρησιμοποιείται η μονάδα reg. Η περιγραφή συμπεριφοράς για να γίνει inference του καταχωρητή είναι και στις δύο περιπτώσεις πολύ απλή και φαίνεται παρακάτω.

Block Διάγραμμα





VHDL Κώδικας

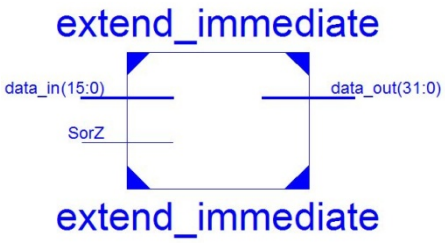
```
library ieee;
use ieee.std_logic_1164.all;

entity reg_we is
    generic(    W         : integer := 32);
    port(   clk           : in  std_logic;
            rst           : in  std_logic;
            we            : in  std_logic;
            data_in       : in  std_logic_vector(W - 1 downto 0);
            data_out      : out std_logic_vector(W - 1 downto 0));
end reg_we;

architecture Behavioral of reg_we is

begin
    process(clk, rst)
    begin

        if(rst = '1') then

            data_out <= (others => '0');

        elsif(clk'event and clk = '1') then

            if(we = '1') then

                data_out <= data_in;

            end if;

        end if;

    end process;
end Behavioral;
```

Extend Immediate

Γενική περιγραφή

Η συνδυαστική αυτή μονάδα είναι υπεύθυνη για την επέκταση προσήμου ή μηδενός στις εντολές immediate ανάλογα με την τιμή του σήματος ελέγχου SorZ. Δέχεται σαν είσοδο τα 16 λιγότερα σημαντικά ψηφία (LSBs) της εντολής immediate και δημιουργεί μια 32 bit έξοδο που έχει για LSBs τα προαναφερθέντα ενώ τα MSBs έχουν είτε την τιμή 0 αν το σήμα ελέγχου SorZ είναι 0, είτε την τιμή του προσήμου (MSB) της εισόδου των 16 bits αν το σήμα ελέγχου είναι 1.

Block διάγραμμα



VHDL κώδικας

```
library ieee;
use ieee.std_logic_1164.all;

entity extend_immediate is
    port(   data_in     : in  std_logic_vector(15 downto 0);
            SorZ        : in  std_logic;
            data_out    : out std_logic_vector(31 downto 0));
end extend_immediate;

architecture Structural of extend_immediate is

begin

    data_out(15 downto 0)   <= data_in;
    data_out(31 downto 16)  <= (others => (data_in(15) and SorZ));

end Structural;
```

Register File

Γενική περιγραφή

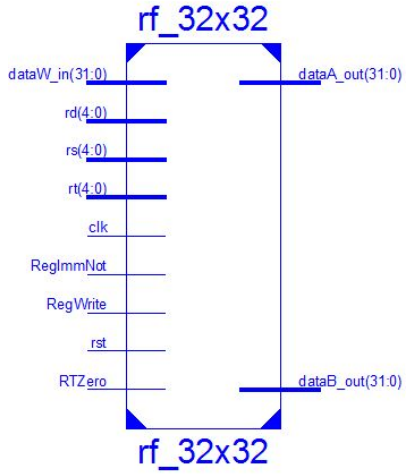
Το αρχείο καταχωρητών περιέχει 32 καταχωρητές γενικής χρήσης των 32 bits ο καθένας. Ο πρώτος από αυτός (R0) έχει μονίμως την τιμή 0 καθώς δε μπορέι να γίνει εγγραφή σε αυτόν. Για την υλοποίηση χρησιμοποιήθηκε περιγραφή συμπεριφοράς η οποία κάνει inference μια κατανεμημένη σε LUTs μνήμη (distributed ram). Εναλλακτικά θα μπορούσαμε να επιλέξουμε να γίνεται inference μιας κανονικής block ram (BRAM) απλά τοποθετώντας τις δηλώσεις εξόδου μέσα στην ακολουθιακή διεργασία (clocked process). Δεν επιλέξαμε να το κάνουμε γιατί θεωρήσαμε πως λόγω του μεγέθους του αρχείου καταχωρητών (32 x 32 = 1KBits) θα ήταν προτιμότερο να μη σπαταλήσουμε μια ολόκληρη BRAM μεγέθους 16 + 2 Kbytes αλλά ανταυτού να χρησιμοποιήσουμε κάποια από τα πολλά διαθέσιμα slices του FPGA. Εάν το σχέδιο μας ήταν

μεγαλύτερο και είχαμε θέμα χώρου ενώ παράλληλα δε μας ένοιαζαν τόσο οι BRAMs, επειδή π.χ χρησιμοποιούσαμε εξωτερική μνήμη SRAM ή SDRAM (DDR) εντολών και δεδομένων, το αντίθετο θα ήταν προτιμότερο.

Τα σήματα ελέγχου που δέχεται η μονάδα είναι τα εξής:

Όνομα	Περιγραφή
RegWrite	Σήμα ενεργοποίησης εγγραφής
RegImmNot	Επιλογή καταχωρητή εγγραφής ανάλογα την εντολή (Register ή Immediate)
RTZero	Θέτει ως δεύτερο καταχωρητή ανάγνωσης τον R0

Block διάγραμμα



VHDL κώδικας

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity rf_32x32 is
    port(   clk         : in  std_logic;
            rst         : in  std_logic;
            RegWrite    : in  std_logic;
            RegImmNot   : in  std_logic;
            RTZero      : in  std_logic;
            rs          : in  std_logic_vector(4 downto 0);
            rt          : in  std_logic_vector(4 downto 0);
            rd          : in  std_logic_vector(4 downto 0);
            dataW_in    : in  std_logic_vector(31 downto 0);
            dataA_out   : out std_logic_vector(31 downto 0);
            dataB_out   : out std_logic_vector(31 downto 0));
end rf_32x32;
```

```
architecture Behavioral of rf_32x32 is

    type ram_distr is array (0 to 31) of std_logic_vector(31 downto 0);
    signal regfile  : ram_distr := (others => (others=>'0'));
    signal rd_a     : std_logic_vector(4 downto 0);
    signal rs_a     : std_logic_vector(4 downto 0);
    signal rt_a     : std_logic_vector(4 downto 0);

begin

    rd_a    <= rd when RegImmNot = '1' else rt;
    rs_a    <= rs;
    rt_a    <= (others => '0') when RTZero = '1' else rt;

    process(clk, rst, RegWrite, rd_a, rs_a, rt_a, dataW_in, regfile)
    begin

        if (rst = '1') then
            dataA_out <= (others => '0');
            dataB_out <= (others => '0');
        else
            -- Single Port Write (Synchronous)
            if(rising_edge(clk))then
                if(RegWrite = '1' and rd_a /= "00000") then
                    regfile(to_integer(unsigned(rd_a))) <= dataW_in;
                end if;
            end if;

            -- Dual Port Read (Asynchronous, infers distributed ram)
            dataA_out <= regfile(to_integer(unsigned(rs_a)));
            dataB_out <= regfile(to_integer(unsigned(rt_a)));
        end if;

    end process;

end Behavioral;
```

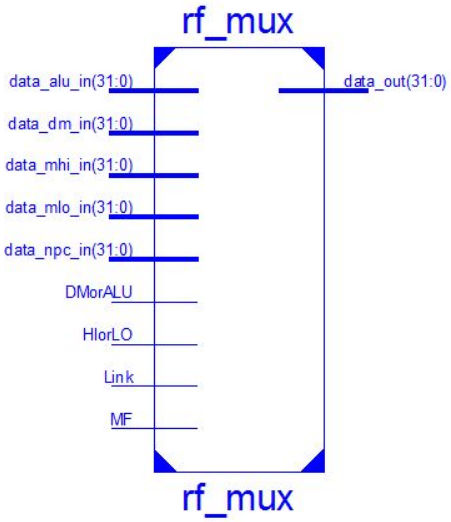
Register Write Multiplexer

Γενική περιγραφή

Η συνδυαστική αυτή μονάδα υλοποιεί έναν πολυπλέκτη 5 πρός 1 εύρους 32 bit ο οποίος έχει σαν έξοδο μια από τις 5 εισόδους ανάλογα με τη τιμή των σημάτων ελέγχου που δέχεται από τη μονάδα ελέγχου. Οι είσοδοι αυτοί είναι οι καταχωρητές NPC, ALUOUT, HI και LO οι οποίοι περιγράφτηκαν σε προηγούμενη ενότητα καθώς και η λέξη που πιθανώς φορτώθηκε από τη μνήμη δεδομένων εαν εκτελείται εντελή φόρτωσης δεδομένων. Η έξοδος οδηγείται στο αρχείο καταχωρητών πρός εγγραφή εφόσον η λειτουργία της εντολής απαιτεί εγγραφή σε κάποιον από αυτούς. Τα σήματα ελέγχου φαίνονται παρακάτω.

Όνομα	Περιγραφή
Link	Εντολές που κάνουν “Link” όπως οι JAL και JALR
DMorALU	Εντολές φόρτωσης δεδομένων όπως οι LW, LH, LHU, LB, και LBU
MF	Εντολές αποθήκευσης της τιμής των καταχωρητών HI ή LO
HIorLO	Επιλογή του καταχωρητή HI ή LO

Block διάγραμμα



VHDL κώδικας

```
library ieee;
use ieee.std_logic_1164.all;

entity rf_mux is
    port(   data_alu_in : in  std_logic_vector(31 downto 0);
            data_dm_in  : in  std_logic_vector(31 downto 0);
            data_npc_in : in  std_logic_vector(31 downto 0);
            data_mlo_in : in  std_logic_vector(31 downto 0);
            data_mhi_in : in  std_logic_vector(31 downto 0);
            Link        : in  std_logic;
            DMorALU     : in  std_logic;
            MF          : in  std_logic;
            HIorLO      : in  std_logic;
            data_out    : out std_logic_vector(31 downto 0));
end rf_mux;

architecture Behavioral of rf_mux is

begin

    process(Link, DMorALU, MF, HIorLO, data_alu_in, data_dm_in, data_mhi_in,
data_mlo_in, data_npc_in)
    begin
        if(Link = '0' and MF = '0' and DMorALU = '0') then      -- Write
register from ALU
            data_out <= data_alu_in;
        elsif(Link = '0' and MF = '0' and DMorALU = '1') then   -- Write
register from DM
            data_out <= data_dm_in;
        elsif(Link = '0' and MF = '1' and HIorLO = '1') then    -- Write
register from HI
            data_out <= data_mhi_in;
        elsif(Link = '0' and MF = '1' and HIorLO = '0') then    -- Write
register from LO
            data_out <= data_mlo_in;
        elsif(Link = '1') then                                  -- Write
register from NPC
            data_out <= data_npc_in;
        else
```

```
            data_out <= (others => '-');
        end if;
    end process;

end Behavioral;
```

Data Memory Control

Γενική περιγραφή

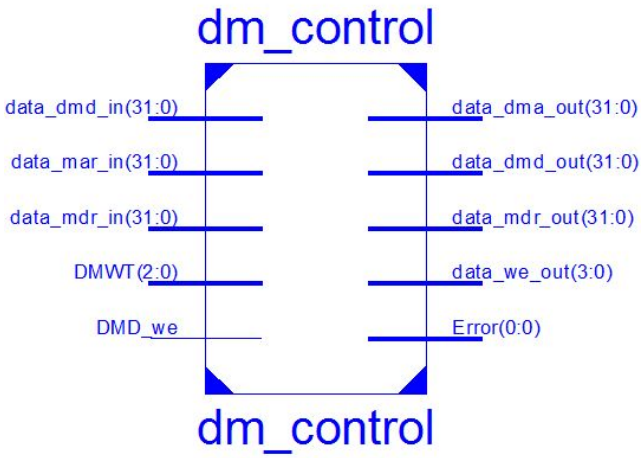
Η συνδυαστική αυτή μονάδα είναι υπεύθυνη για τη φόρτωση ή αποθήκευση μιας λέξης από / πρός τη μνήμη δεδομένων. Η υλοποίηση της περιλαμβάνει τρία κομμάτια τα οποία είναι υπεύθυνα για τον έλεγχο ευθυγράμμισης, τον καθορισμό του μήκους της λέξης και την φόρτωση ή αποθήκευση αυτής.

Το πρώτο κομμάτι που αφορά τον έλεγχο ευθυγράμμισης ελέγχει τον τύπο της εντολής από το σήμα DMWT (Date Memory Word Type) των 3bits καθώς και τα 2 LSBs της διεύθυνσης. Εαν αυτά δε συμβαδίζουν τότε το σήμα εξόδου Error αποκτά τη λογική τιμή 1. Αυτό συμβαίνει είτε όταν έχουμε εντολή πλήρους λέξης (LW, SW) και τα LSBs της διεύθυνσης δεν είναι 00 είτε όταν έχουμε εντολή μισής λέξης (LH, LHU, SH) και τα LSBs είναι 01 ή 11.

Το επόμενο κομμάτι φροντίζει για τον καθορισμό του μήκους της λέξης που πρόκειται να φορτωθεί ή αποθηκευτεί ανάλογα με τον τύπο της εντολής και τα 2 LSBs της διεύθυνσης. Επίσης θέτει τα κατάλληλα σήματα ενεργοποίησης εγγραφής της μνήμης ώστε να μην αλλάξουν στη μνήμη bytes που δεν πρέπει με βάση πάλι τον τύπο της εντολής και τα LSBs της διεύθυνσης.

Το τρίτο κομμάτι είναι πολύ απλό και απλά κάνει την φόρτωση ή την αποθήκευση της λέξης που καθορίστηκε στο προηγούμενο κομμάτι κώδικα. Όπως είναι ανεμενόμενο εαν το σήμα ελέγχου DMD_we είναι 0 εκτελεί ανάγνωση ενώ αν είναι 1 εκτελεί εγγραφή.

b



VHDL κώδικας

```
library ieee;
use ieee.std_logic_1164.all;

entity dm_control is
    port(   data_mdr_in     : in  std_logic_vector(31 downto 0);
            data_mar_in     : in  std_logic_vector(31 downto 0);
            data_dmd_in     : in  std_logic_vector(31 downto 0);
            DMWT            : in  std_logic_vector(2 downto 0);
            DMD_we          : in  std_logic;
            Error           : out std_logic_vector(0 downto 0);
            data_mdr_out    : out std_logic_vector(31 downto 0);
            data_dma_out    : out std_logic_vector(31 downto 0);
            data_we_out     : out std_logic_vector(3 downto 0);
            data_dmd_out    : out std_logic_vector(31 downto 0));
end dm_control;

architecture Behavioral of dm_control is

    signal lsbits       : std_logic_vector(1 downto 0);
    signal data_read    : std_logic_vector(31 downto 0);
    signal data_write   : std_logic_vector(31 downto 0);
    signal data_we      : std_logic_vector(3 downto 0);
    signal data_error   : std_logic;

begin

    lsbits      <= data_mar_in(1 downto 0);
    Error(0)    <= data_error;

    process(DMWT, lsbits, data_mar_in)
    begin

        -- Check address alignment

        if(DMWT(2) = '1' and  lsbits /= "00") then                          --
Unaligned LW, SW
            data_dma_out    <= (others => '0');
            data_error      <= '1';
        elsif(DMWT(1) = '1' and (lsbits = "01" or lsbits = "11")) then    --
Unaligned LH, LHU, SH
            data_dma_out    <= (others => '0');
            data_error      <= '1';
        else
            data_dma_out    <= data_mar_in;
            data_error      <= '0';                                         --
LB, LBU, SB and All aligned LW, SW, LH, LHU, SH
        end if;

    end process;

    process(DMWT, lsbits, data_dmd_in, data_mdr_in, data_we)
    begin

        data_read   <= (others => '-');
        data_write  <= (others => '-');
        data_we     <= "0000";

        -- Read or write the correct bytes

        if(DMWT(2) = '1') then      -- LW, SW
            data_read   <= data_dmd_in;
```

```
            data_write  <= data_mdr_in;
            data_we     <= "1111";
        elsif(DMWT(1) = '1') then   -- LH, LHU, SH
            if(lsbits = "00") then
                data_read(15 downto 0)      <= data_dmd_in(15 downto 0);
                data_read(31 downto 16)     <= (others => (data_dmd_in(15) and
DMWT(0)));

                data_write(15 downto 0) <= data_mdr_in(15 downto 0);
                data_write(31 downto 16)    <= (others => (data_mdr_in(15) and
DMWT(0)));
                data_we                     <= "0011";
            elsif(lsbits = "10") then
                data_read(15 downto 0)      <= data_dmd_in(31 downto 16);
                data_read(31 downto 16)     <= (others => (data_dmd_in(31) and
DMWT(0)));

                data_write(15 downto 0)     <= data_mdr_in(31 downto 16);
                data_write(31 downto 16)    <= (others => (data_mdr_in(31) and
DMWT(0)));
                data_we                     <= "1100";
            end if;

        elsif(DMWT(1) = '0') then   -- LB, LBU, SB
            if(lsbits = "00") then
                data_read(7 downto 0)   <= data_dmd_in(7 downto 0);
                data_read(31 downto 8)  <= (others => (data_dmd_in(7) and
DMWT(0)));

                data_write(7 downto 0)  <= data_mdr_in(7 downto 0);
                data_write(31 downto 8) <= (others => '0');
                data_we                 <= "0001";

            elsif(lsbits = "01") then
                data_read(7 downto 0)   <= data_dmd_in(15 downto 8);
                data_read(31 downto 8)  <= (others => (data_dmd_in(15) and
DMWT(0)));

                data_write(7 downto 0)  <= (others => '0');
                data_write(15 downto 8) <= data_mdr_in(7 downto 0);
                data_write(31 downto 16) <= (others => '0');
                data_we                 <= "0010";
            elsif(lsbits = "10") then
                data_read(7 downto 0)   <= data_dmd_in(23 downto 16);
                data_read(31 downto 8)  <= (others => (data_dmd_in(23) and
DMWT(0)));

                data_write(15 downto 0) <= (others => '0');
                data_write(23 downto 16) <= data_mdr_in(7 downto 0);
                data_write(31 downto 24) <= (others => '0');
                data_we                  <= "0100";
            elsif(lsbits = "11") then
                data_read(7 downto 0)   <= data_dmd_in(31 downto 24);
                data_read(31 downto 8)  <= (others => (data_dmd_in(31) and
DMWT(0)));

                data_write(23 downto 0) <= (others => '0');
                data_write(31 downto 24) <= data_mdr_in(7 downto 0);
                data_we                 <= "1000";
            end if;
        end if;
    end process;
```

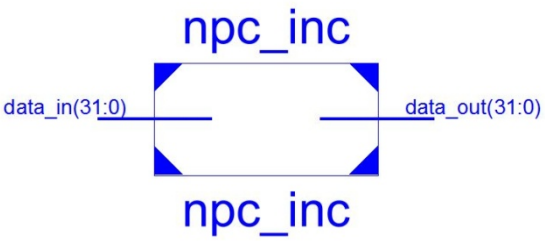
```
    process(DMD_we, data_write, data_read, data_we, data_error)
    begin

        -- Load or store the data (do nothing on error error)

        if(DMD_we = '1' and data_error = '0') then
            data_we_out     <= data_we;
            data_dmd_out    <= data_write;
            data_mdr_out    <= (others => '-');
        elsif(DMD_we = '0' and data_error = '0') then
            data_we_out     <= (others => '0');
            data_dmd_out    <= (others => '-');
            data_mdr_out    <= data_read;
        else
            data_we_out     <= (others => '0');
            data_dmd_out    <= (others => '-');
            data_mdr_out    <= (others => '-');
        end if;

    end process;

end Behavioral;
```

NPC Incrementer

Γενική περιγραφή

Η συνδυαστική αυτή μονάδα περιλαμβάνει έναν μη προσημασμένο αθροιστή των 32 bits ο οποίος δέχεται σαν εισόδους την τιμή του καταχωρητή PC (Program Counter) και τον σταθερό αριθμό 4 ώστε να δημιουργηθεί στην έξοδο της η νέα τιμή του καταχωρητή PC για την επόμενη εντολή εφόσον δεν εκτελείται μια εντολή που αλλάζει τη ροή του προγράμματος (Branch ή Jump)

Block διάγραμμα



VHDL κώδικας

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity npc_inc is
    port(   data_in     : in  std_logic_vector(31 downto 0);
            data_out    : out std_logic_vector(31 downto 0));
end npc_inc;
```

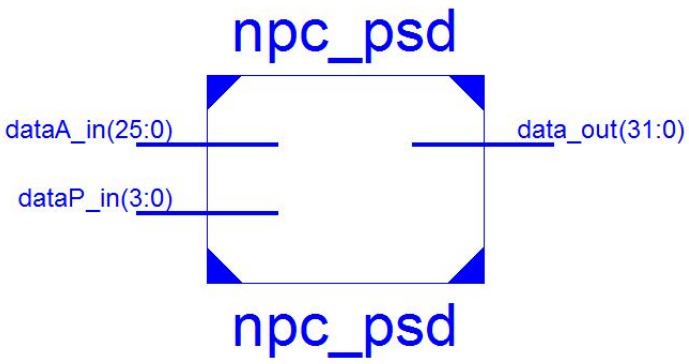
```vhdl
architecture Structural of npc_inc is

    constant four : unsigned(31 downto 0) := (2 => '1', others => '0');

begin

    data_out <= std_logic_vector(unsigned(data_in) + four);

end Structural;
```

NPC PSD

Γενική περιγραφή

Η συνδυαστική αυτή μονάδα δέχεται σαν είσοδους τα 4 MSBs του καταχωρητή PC και τα 26 LSBs της εντολής που εκτελείται (όλα τα bits εκτός από το opcode) βγάζοντας στην έξοδο τη συνένωση αυτών μαζί με 2 μηδενικά στις τελευταίες θέσεις (4 + 26 + 2 = 32 bits). Η τιμή αυτή που παράγεται θα χρησιμοποιηθεί για την αλλαγή της ροής του προγράμματος σε περίπτωση που έχουμε εντολή Jump και αποθηκεύεται προσωρινά στον καταχωρητή D ώστε να είναι διαθέσιμη στον NPC Mux.

Block διάγραμμα



VHDL κώδικας

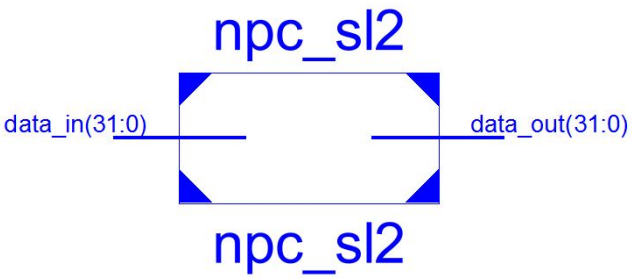
```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity npc_psd is
    port(   dataP_in    : in  std_logic_vector(3 downto 0);
            dataA_in    : in  std_logic_vector(25 downto 0);
            data_out    : out std_logic_vector(31 downto 0));
end npc_psd;

architecture Structural of npc_psd is

begin

    data_out <= dataP_in & dataA_in & "00";

end Structural;
```

NPC Shifter

Γενική περιγραφή

Η συνδυαστική αυτή μονάδα κάνει ολίσθηση κατά δύο θέσης του πεδίου immediate. Η έξοδος καταλήγει στον NPC Adder ο οποίος με τη σειρά του τη δίνει στον NPC Mux ώστε να χρησιμοποιηθεί στην περίπτωση που η εντολή είναι τύπου Branch.

Block διάγραμμα



VHDL κώδικας

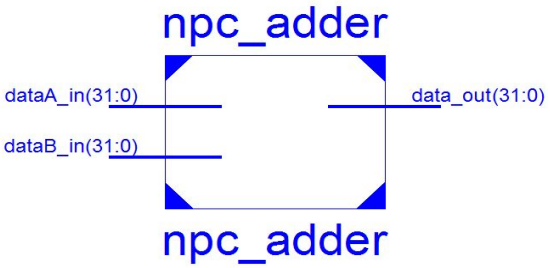
```
library ieee;
use ieee.std_logic_1164.all;

entity npc_sl2 is
    port(   data_in     : in  std_logic_vector(31 downto 0);
            data_out    : out std_logic_vector(31 downto 0));
end npc_sl2;

architecture Structural of npc_sl2 is

begin

    data_out <= data_in(29 downto 0) & "00";

end Structural;
```

NPC Adder

Γενική περιγραφή

Η συνδυαστική αυτή μονάδα περιλαμβάνει έναν μη προσημασμένο αθροιστή των 32 bits ο οποίος δέχεται σαν εισόδους την τιμή του καταχωρητή A που μόλις φορτώθηκε από το αρχείο καταχωρητών και την τιμή του πεδίου immediate ολισθημένη κατά 2 θέσεις αριστερά από τη μονάδα NPC Shifter. Η έξοδος αποθηκεύεται στον καταχωρητή M ο οποίος τελικά καταλήγει στον NPC Mux ώστε να χρησιμοποιηθεί στην περίπτωση που η εντολή είναι τύπου Branch.

Block διάγραμμα



VHDL κώδικας

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity npc_adder is
    port(   dataA_in    : in  std_logic_vector(31 downto 0);
            dataB_in    : in  std_logic_vector(31 downto 0);
            data_out    : out std_logic_vector(31 downto 0));
end npc_adder;

architecture Structural of npc_adder is

begin
    data_out <= std_logic_vector(unsigned(dataA_in) + unsigned(dataB_in));
end Structural;
```

NPC Selector

Γενική περιγραφή

Η συνδυαστική αυτή μονάδα έχει σα στόχο την παραγωγή του σήματος επιλογής της μονάδας NPC Mux. Η παραγωγή του σήματος αυτού από τη συγκεκριμένη μονάδα αντί από τη κεντρική μονάδα ελέγχου έχει το πλεονέκτημα της απλοποίησης της κεντρικής μονάδας ελέγχου και των σημάτων που αυτή παράγει εκμεταλευόμενη τις διάφορες λογικές πράξεις μεταξύ των σημάτων ελέγχου που δέχεται. Τα σήματα αυτά φαίνονται παρακάτω.

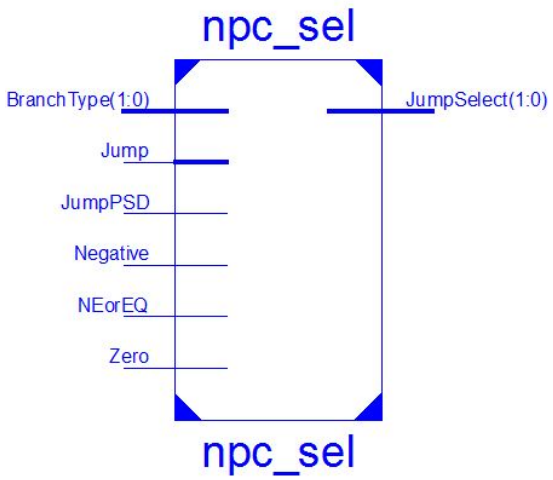
Όνομα	Περιγραφή
BranchType	Καθορίζει αν έχουμε κανονική εντολή ή κάποια τύπου Branch
Jump	Καθορίζει αν κάνουμε Jump με τις εντολές J, JAL, JR, JALR
JumpPSD	Καθορίζει αν κάνουμε Jump με τις εντολές J, JAL
Zero	Το αποτέλεσμα της ALU είναι μηδέν
Negative	Το αποτέλεσμα της ALU είναι μικρότερο του μηδενός
NEorEQ	Καθορίζει ποια από τις δύο εντολές εκτελείται σε κάθε ζευγάρι εντολών Branch: BEQ ή BNE, BLEZ ή BGTZ, BLTZ ή BGEZ

Τα τρία τελευταία σήματα συνδυάζονται με λογικές πράξεις ώστε να καθοριστεί ο τύπος του Branch που εκτελείται και αν αυτό είναι επιτυχημένο ή οχι. Ενδιαφέρον παρουσιάζει η περίπτωση των BLEZ / BGTZ branches όπου το αν είναι επιτυχημένο καθορίζεται από ένα Shannon expansion και των τριών σημάτων ώς εξής:

BranchTaken = ((not NEorEQ) and (Zero or Negative)) or ((NEorEQ) and (Zero nor Negative))

Αυτός ο τρόπος υπολογισμού μας απαλάσει από την υλοποίηση ενός μεγαλύτερου πολυπλέκτη ο οποίος είναι ακριβή και μη αποδοτική λύση στα περισσότερα FPGA.

Block διάγραμμα



VHDL κώδικας

```
library ieee;
use ieee.std_logic_1164.all;

entity npc_sel is
    port(   Jump        : in  std_logic;
            JumpPSD     : in  std_logic;
            BranchType  : in  std_logic_vector(1 downto 0);
            NEorEQ      : in  std_logic;
            Zero        : in  std_logic;
            Negative    : in  std_logic;
            JumpSelect  : out std_logic_vector(1 downto 0));
end npc_sel;

architecture Behavioral of npc_sel is
    signal BranchSelect : std_logic;
begin

    process(BranchType, Zero, NEorEQ, Negative)
    begin

        if(BranchType = "00") then              -- Sequential
            BranchSelect <= '0';
        elsif(BranchType = "01") then           -- BEQ, BNE
            BranchSelect <= Zero xor NEorEQ;
        elsif(BranchType = "10") then           -- BLEZ, BGTZ
            BranchSelect <= ((not NEorEQ) and (Zero or Negative)) or ((NEorEQ)
and (Zero nor Negative));
```

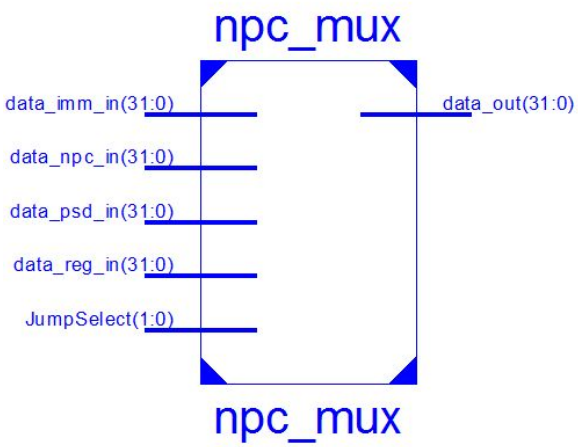
```
        elsif(BranchType = "11") then           -- BLTZ, BGEZ
            BranchSelect <= Negative xor NEorEQ;
        end if;

    end process;

    JumpSelect(1) <= Jump;

    JumpSelect(0) <= BranchSelect when Jump = '0' else
                     JumpPSD when Jump = '1' else
                     '0';

end Behavioral;
```

NPC Multiplexer

Γενική περιγραφή

Η συνδυαστική αυτή μονάδα υλοποιεί έναν πολυπλέκτη 4 πρός 1 εύρους 32 bit ο οποίος έχει σαν έξοδο μια από τις 4 εισόδους ανάλογα με τη τιμή του σήματος ελέγχου που δέχεται από τη μονάδα NPC Selector. Οι είσοδοι είναι οι καταχωρητές A, M, NPC και D οι οποίοι περιγράφτηκαν σε προηγούμενη ενότητα. Η έξοδος οδηγείται στον καταχωρητή PC ώστε στον επόμενο κύκλο εκτέλεσης να φορτωθεί η κατάλληλη εντολή από τη μνήμη εντολών.

Block διάγραμμα



VHDL κώδικας

```
library ieee;
use ieee.std_logic_1164.all;

entity npc_mux is
    port(   data_npc_in : in  std_logic_vector(31 downto 0);
            data_imm_in : in  std_logic_vector(31 downto 0);
            data_reg_in : in  std_logic_vector(31 downto 0);
            data_psd_in : in  std_logic_vector(31 downto 0);
            JumpSelect  : in  std_logic_vector(1 downto 0);
            data_out    : out std_logic_vector(31 downto 0));
end npc_mux;
```

```
architecture Structural of npc_mux is

begin

    with JumpSelect select

        data_out    <=  data_npc_in when "00",          -- Sequential
                        data_imm_in when "01",          -- Branch
                        data_reg_in when "10",          -- JR and JALR
                        data_psd_in when "11",          -- J  and JAL
                        (others => '-') when others;
end Structural;
```

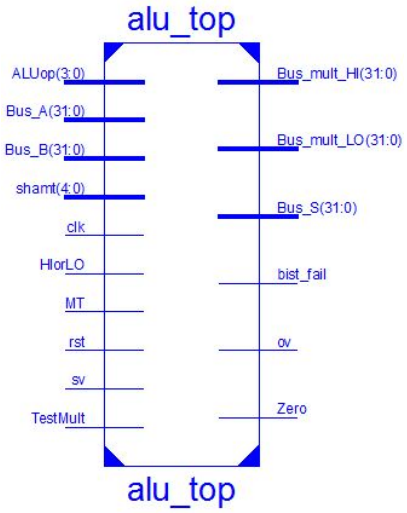
ALU Top

Γενική περιγραφή

Η μονάδα αυτή εκτελεί όλες τις αριθμητικές και λογικές πράξεις απευθείας ενώ περιλαμβάνει και υπομονάδες για την εκτέλεση της πράξης του πολλαπλασιασμού καθώς και των πράξεων ολίσθησης όλων των μορφών (λογικές / αριθμητικές, προς τα αριστερά ή προς τα δεξιά) ακόμα και αυτή που απαιτεί η εντολή LUI (hardcoded αριστερή ολίσθηση κατά 16 θέσεις). Περιλαμβάνει επίσης κύκλωμα ανίχνευσης του μηδενός το οποίο κατά τη σύνθεση γίνεται ουσιαστικά ένα δένδρο OR. Τέλος περιλαμβάνει αρκετές δηλώσεις επιλογής ή πολυπλέκτες που επιλέγουν το μέγεθος της ολίσθησης, την εγγραφή των καταχωρητών HI και LO, την πράξη που εκτελείται, την επιθυμητή έξοδο στον καταχωρητή ALUOUT καθώς και τα σήματα Zero, Oveflow. Τα σήματα ελέγχου που δέχεται σαν είσοδο η μονάδα είναι τα εξής;

Όνομα	Περιγραφή
TestMult	Ειδοποιεί το multiplier unit να ξεκινήσει τη διαδικασία αυτοελέγχου (BIST)
ALUop	Καθορίζει τη πράξη που θα εκτελεστεί
shamt	Το ποσό της ολίσθησης
sv	Καθορίζει αν εκτελούμε ολίσθηση σταθερού ή μεταβλητού μεγέθούς
MT	Καθορίζει αν εκτελούμε εντολή εγγραφής των καταχωρητών HI ή LO
HIorLO	Ο τύπος της εγγραφής, 1 αν θέλουμε να εγγραφεί ο HI και 0 για τον LO

Block διάγραμμα



VHDL κώδικας

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity alu_top is
    generic(mult_pipe   : boolean := true);
    port(   clk         : in  std_logic;
            rst         : in  std_logic;
            sv          : in  std_logic;
            TestMult    : in  std_logic;
            MT          : in  std_logic;
            HIorLO      : in  std_logic;
            ALUop       : in  std_logic_vector(3 downto 0);
            shamt       : in  std_logic_vector(4 downto 0);
            Bus_A       : in  std_logic_vector(31 downto 0);
            Bus_B       : in  std_logic_vector(31 downto 0);
            Zero        : out std_logic;
            ov          : out std_logic;
            bist_fail   : out std_logic;
            Bus_S       : out std_logic_vector(31 downto 0);
            Bus_mult_HI : out std_logic_vector(31 downto 0);
            Bus_mult_LO : out std_logic_vector(31 downto 0));
end alu_top;

architecture Behavioral of alu_top is

    component alu_shifter is
        port(   left        : in  std_logic;
                logical     : in  std_logic;
                shift       : in  std_logic_vector(4 downto 0);
                shift_in    : in  std_logic_vector(31 downto 0);
                shift_out   : out std_logic_vector(31 downto 0));
    end component;

    component alu_mult_top is
        generic(mult_pipe   : boolean := true);
        port(   clk         : in  std_logic;
                rst         : in  std_logic;
                bist_init   : in  std_logic;
                X           : in  std_logic_vector(31 downto 0);
                Y           : in  std_logic_vector(31 downto 0);
                P_HI        : out std_logic_vector(31 downto 0);
                P_LO        : out std_logic_vector(31 downto 0);
                bist_fail   : out std_logic);
    end component;

    signal tmp_result_hi    : std_logic_vector(31 downto 0);
    signal tmp_result_lo    : std_logic_vector(31 downto 0);
    signal L_out            : std_logic_vector(31 downto 0);
    signal A_out            : std_logic_vector(31 downto 0);
    signal Sh_out           : std_logic_vector(31 downto 0);
    signal SLT_out          : std_logic_vector(31 downto 0);
    signal output           : std_logic_vector(31 downto 0);
    signal shift            : std_logic_vector(4 downto 0);
    signal left             : std_logic;
    signal logical          : std_logic;

begin
```

```
    MULT : alu_mult_top
    generic map(mult_pipe   => mult_pipe)
    port map(   clk         => clk,
                rst         => rst,
                bist_init   => TestMult,
                X           => Bus_A,
                Y           => Bus_B,
                P_HI        => tmp_result_hi,
                P_LO        => tmp_result_lo,
                bist_fail   => bist_fail);

    SHIFTER : alu_shifter
    port map(   left        => left,
                logical     => logical,
                shift       => shift,
                shift_in    => Bus_B,
                shift_out   => output);

    shift       <= Bus_A(4 downto 0) when (sv = '1') else shamt;
    Bus_mult_HI <= tmp_result_hi when (ALUop(1 downto 0) = "00" and MT = '0')
else Bus_A when (MT = '1' and HIorLO = '1') else (others => '-');
    Bus_mult_LO <= tmp_result_lo when (ALUop(1 downto 0) = "00" and MT = '0')
else Bus_A when (MT = '1' and HIorLO = '0') else (others => '-');
    Zero        <= '1' when (A_out = X"00000000") else '0';

    process(Bus_A, Bus_B, ALUop, output)
        variable tmp_add_sub: std_logic_vector(32 downto 0);
    begin

        ov      <= '0';
        SLT_out <= (others=>'-');
        A_out   <= (others=>'-');

        -- Shift
        left    <='0';
        logical <='1';

        case ALUop(1 downto 0) is

            when "00" =>

                tmp_add_sub := std_logic_vector(signed(Bus_A(31) & Bus_A) +
signed(Bus_B(31) & Bus_B));

                Sh_out      <= output;
                A_out       <= tmp_add_sub(31 downto 0);
                L_out       <= Bus_A and Bus_B;

                left        <='1';
                ov          <= (Bus_A(31) and Bus_B(31) and (not A_out(31)))
                                or ((not Bus_A(31)) and (not Bus_B(31)) and
A_out(31));

            --Truncate 2 MSBits
            when "01" =>

                tmp_add_sub := std_logic_vector(unsigned('0' & Bus_A) +
unsigned('0' & Bus_B));

                Sh_out      <= (others => '-');
                A_out       <= tmp_add_sub(31 downto 0);
                L_out       <= Bus_A or Bus_B;
```

```
                ov          <= tmp_add_sub(32);

            when "10" =>

                tmp_add_sub := std_logic_vector(signed(Bus_A(31) & Bus_A) -
signed(Bus_B(31) & Bus_B));

                if((Bus_A(31) xor Bus_B(31)) = '1') then
                    SLT_out <= "000" & X"0000000" & Bus_A(31);
                else
                    SLT_out <= "000" & X"0000000" & tmp_add_sub(31);
                end if;

                Sh_out      <= output;
                A_out       <= tmp_add_sub(31 downto 0);
                L_out       <= Bus_A xor Bus_B;

                ov          <= ((not Bus_A(31)) and Bus_B(31) and A_out(31))
                               or (Bus_A(31) and (not Bus_B(31)) and (not
A_out(31)));

            when "11" =>

                tmp_add_sub := std_logic_vector(unsigned('0' & Bus_A) -
unsigned('0' & Bus_B));

                Sh_out      <= output;
                SLT_out     <= "000" & X"0000000" & tmp_add_sub(32);
                A_out       <= tmp_add_sub(31 downto 0);
                L_out       <= Bus_A nor Bus_B;

                logical     <= '0';
                ov          <= tmp_add_sub(32);

            when others =>

                L_out       <= (others=>'-');

        end case;

    end process;

    process(Sh_out, SLT_out, A_out, L_out, ALUop(3 downto 2))
    begin
        -- Mux
        case ALUop(3 downto 2) is
            when "00"   =>  Bus_S <= Sh_out;    -- Shift
            when "01"   =>  Bus_S <= SLT_out;   -- SLT
            when "10"   =>  Bus_S <= A_out;     -- Arithmetic
            when "11"   =>  Bus_S <= L_out;     -- Logical
            when others =>  Bus_S <= (others => '-');
        end case;
    end process;

end Behavioral;
```

ALU Multiplier

Γενική περιγραφή

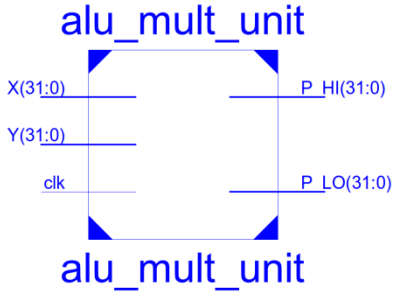
Η συγκεκριμένη μονάδα περιλαμβάνει αρκετές υπομονάδες οι περισσότερες των οποίων είναι χρήσιμες μόνο στην περίπτωση που εκτελούμε την εντολή TEST η οποία εκκινεί τις τρείς τεχνικές αυτοδοκιμής του πολλαπλασιαστή και οι οποίες περιγράφονται σε επόμενες ενότητες. Στην περιγραφή μας εδώ θα εστιάσουμε μόνο στην μονάδα του πολλαπλασιαστή και όχι στις υπόλοιπες.

Η μονάδα του πολλαπλασιαστή εκτελεί την πράξη του πολλαπλασιασμού μεταξύ δύο προσημασμένων 32bit αριθμών και παράγει στην έξοδο το 64 bit αποτέλεσμα. Περιλαμβάνει δύο είδη πολλαπλασιαστών όπου η επιλογή του καθενώς για σύνθεση γίνεται μέσω μιας generic boolean παραμέτρου. Εαν η συγκεκριμένη παράμετρος έιναι αληθής τότε δημιουργείται ενας ακολουθιακός pipelined πολλαπλασιαστής που παράγει αποτέλεσμα σε 4 κύκλους ρολογιού. Εναλλακτικά αν θέλουμε να μειώσουμε τους κύκλους αυτους σε έναν μπορούμε να θέσουμε την παράμετρο σε ψευδή τιμή όπου σε αυτή την περίπτωση δημιουργείται ένας συνδυαστικός κανονικός πολλαπλασιαστής με καθυστέρηση ενός κύκλου ρολογιού.

Η επιλογή της μορφής του πολλαπλασιαστή επηρεάζει άμεσα το μέγεθος του κρίσιμου μονοπατιού του επεξεργαστή και αρα τη συχνότητα λειτουργίας του. Η επιλογή ενός pipelined πολλαπλασιαστή επιτρέπει τη μείωση του κρίσιμου μονοπατιού και την αύξηση της συχνότητας λειτουργίας όμως έχει το μειονέκτημα οτι χρειάζεται 3 κύκλους παραπάνω για να παράξει το αποτέλεσμα. Η πράξη του πολλαπλασιασμού γενικά θεωρείται μια ακριβή πράξη από τις περισσότερες αρχιτεκτονικές, λόγω οτι και αυτές περιλαμβάνουν τέτοιου είδους πολλαπλασιαστή, οπότε κρίναμε και εμείς οτι αυτή είναι η σωστή προσέγγιση ώστε να εκτελούνται όλες οι υπόλοιπες εντολές πιο γρήγορα. Σε περίπτωση που ο χρήστης κάνει στα προγράμματα του υπερβολικά μεγάλη χρήση της πράξης του πολλαπλασιασμου θα πρέπει να εξετάσει αν έχει νόημα η αλλαγή της παραμέτρου και η επιστροφή στον κανονικό πολλαπλασιαστη του ενός κύκλου.

Ο pipelined πολλαπλασιαστής έχει περιγραφεί με έναν κάπως περίεργο τρόπο τον οποίο βρήκαμε στα εγχειρίδια του συνθέτη της Xilinx (XST). Παρατηρούμε οτι έχουμε μια δήλώση πολλαπλασιαστή και το αποτέλεσμα περνάει από μια σειρά από 4 registers. Στην πραγματικότητα ο συνθέτης δε τοποθετεί απλά 4 registers μετά από τον πολλαπλασιαστή και αυτό είναι απλά ένα template για την καθοδήγηση του συνθέτη από εμας ώστε να παράξει pipelined πολλαπλασιαστή. Ο αριθμός των registers που δηλώνουμε μπορεί να αλλάξει είτε σε λιγότερους (π.χ 3) είτε σε περισσότερους (π.χ 5). Μετά από δοκιμές διαπιστώσαμε οτι το ιδανικό κρίσιμο μονοπάτι δημιουργείται αν τοποθετήσουμε 4 registers στο τέλος για συνολική καθυστέρηση 4 κύκλων.

VHDL Κώδικας



VHDL κώδικας

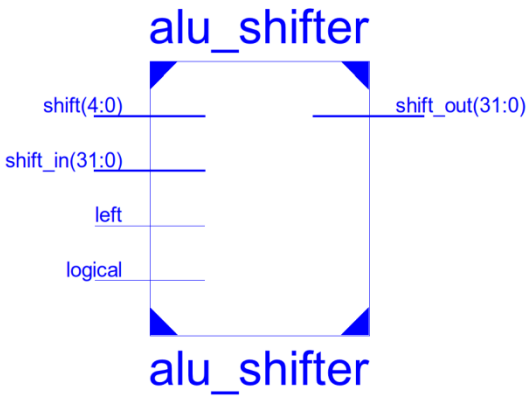
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity alu_mult_unit is
    generic(mult_pipe    : boolean := true);
    port(   clk          : in std_logic;
            X            : in std_logic_vector(31 downto 0);
            Y            : in std_logic_vector(31 downto 0);
            P_HI         : out std_logic_vector(31 downto 0);
            P_LO         : out std_logic_vector(31 downto 0));
            attribute mult_style: string;
            attribute mult_style of alu_mult_unit: entity is "pipe_lut";
end alu_mult_unit;

architecture Structural of alu_mult_unit is
    signal X_signed     : signed(31 downto 0);
    signal X_signed_r1  : signed(31 downto 0);
    signal Y_signed     : signed(31 downto 0);
    signal Y_signed_r1  : signed(31 downto 0);
    signal P_signed     : signed(63 downto 0);
    signal P_signed_r1  : signed(63 downto 0);
    signal P_signed_r2  : signed(63 downto 0);
    signal P_signed_r3  : signed(63 downto 0);
    signal P_vector     : std_logic_vector(63 downto 0);
begin

    X_signed    <= signed(X);
    Y_signed    <= signed(Y);

    pipelined: if (mult_pipe = true) generate

        -- Pipelined multiplier (4 clock cycles latency)
        process(clk)
        begin
            if(clk'event and clk = '1') then
                X_signed_r1 <= X_signed;
                Y_signed_r1 <= Y_signed;
                P_signed_r1 <= X_signed * Y_signed;
                P_signed_r2 <= P_signed_r1;
                P_signed_r3 <= P_signed_r2;
                P_signed    <= P_signed_r3;
            end if;
        end process;

    end generate;

    normal: if(mult_pipe = false) generate

         -- Normal Multiplier (1 clock cycle latency)
         P_signed <= X_signed * Y_signed;

    end generate;

    -- Convert for output
    P_vector    <= std_logic_vector(P_signed);
    P_HI        <= P_vector(63 downto 32);
    P_LO        <= P_vector(31 downto 0);

end Structural;
```

ALU Shifter

Γενική περιγραφή

Η συνδυαστική αυτή μονάδα εκτελεί λογικές ή αριθμητικές ολισθήσεις προς τα αριστερά ή πρός τα δεξιά. Για την υλοποίηση της χρησιμοποιείται ένας πολυπλέκτης 4 πρός 1 ο οποίος έχει σαν εισόδους τα 4 διαφορετικά σενάρια που μπορούν να προκύψουν. Ο πολυπλέκτης έχει για εισόδους 4 διαφορετικούς ολισθητές που κάνει infer το εργαλείο σύνθεσης από την κατάλληλη περιγραφή συμπεριφοράς μας χρησιμοποιώντας VHDL macros.

Block διάγραμμα



VHDL κώδικας

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity alu_shifter is
    port(   left        : in  std_logic;
            logical     : in  std_logic;
            shift       : in  std_logic_vector(4 downto 0);
            shift_in    : in  std_logic_vector(31 downto 0);
            shift_out   : out std_logic_vector(31 downto 0));
end alu_shifter;

architecture Behavioral of alu_shifter is

begin

    process(shift_in, shift, left, logical)
        variable shift_n    : natural range 0 to 31;
        variable in_s       : signed (31 downto 0);
        variable in_u       : unsigned (31 downto 0);
        variable SEL        : std_logic_vector(1 downto 0);
    begin

        in_s    := signed(shift_in);
        in_u    := unsigned(shift_in);
        SEL     := logical & left;
        shift_n := to_integer(unsigned(shift));

        case SEL is
            when "00"   => shift_out <= std_logic_vector(SHIFT_RIGHT(in_s,
shift_n));
            when "01"   => shift_out <= std_logic_vector(SHIFT_LEFT(in_s,
shift_n));
```

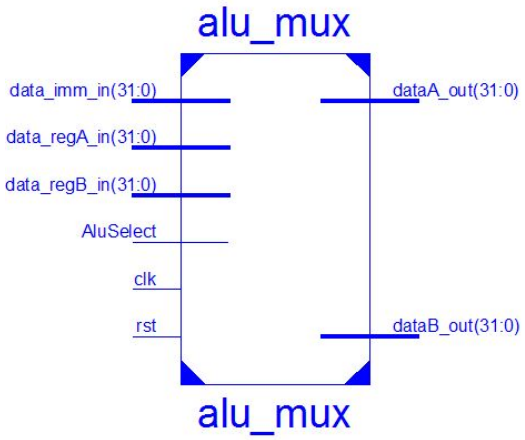
```
            when "10"   => shift_out <= std_logic_vector(SHIFT_RIGHT(in_u,
shift_n));
            when others => shift_out <= std_logic_vector(SHIFT_LEFT(in_u,
shift_n));
        end case;

    end process;

end Behavioral;
```

ALU Multiplexer

Γενική περιγραφή

Η συνδυαστική αυτή μονάδα υλοποιεί έναν πολυπλέκτη 2 πρός 1 εύρους 32 bit ο οποίος έχει σαν έξοδο μια από τις 2 εισόδους ανάλογα με τη τιμή του σήματος ελέγχου που δέχεται από τη μονάδα ελέγχου (BorI). Οι είσοδοι είναι οι καταχωρητές B και I οι οποίοι περιγράφτηκαν σε προηγούμενη ενότητα. Η έξοδος οδηγείται στην ALU

Block διάγραμμα



VHDL κώδικας

```
library ieee;
use ieee.std_logic_1164.all;

entity alu_mux is
    port(   clk             : in  std_logic;
            rst             : in  std_logic;
            data_regA_in    : in  std_logic_vector(31 downto 0);
            data_regB_in    : in  std_logic_vector(31 downto 0);
            data_imm_in     : in  std_logic_vector(31 downto 0);
            AluSelect       : in  std_logic;
            dataA_out       : out std_logic_vector(31 downto 0);
            dataB_out       : out std_logic_vector(31 downto 0));
end alu_mux;

architecture Structural of alu_mux is

begin

    dataA_out <= data_regA_in;
```

```
    with AluSelect select

        dataB_out   <=  data_regB_in    when '1',
                        data_imm_in     when '0',
                        (others => '-') when others;

end Structural;
```

2.2 Τεχνική περιγραφή του multiplier BIST / SBST

Γενική περιγραφή

Ο πολλαπλασιαστής που περιλαμβάνει ο επεξεργαστής μας υποστηρίζει τεχνικές αυτοδοκιμής στο υλικό (BIST) σύμφωνα με τις οποίες μπορεί να ανιχνεύσει λάθη στη λειτουργία του συγκρίνοντας τις τιμές που βγάζει στην έξοδο με αυτές τις οποίες θα έπρεπε να έχει αν δούλευε ορθά. Για να μπορέσει να γίνει η σύγκριση πρέπει τα διανύσματα εισόδου να είναι γνωστά ή να παράγονται με γνωστό τρόπο όπως μέσω κάποιας μαθηματικής ακολουθίας. Επίσης επειδή τα διανύσματα εισόδου μπορεί να θέλουμε (και καλό είναι) να είναι πολλά, για να μη χρειάζεται να αποθηκεύσουμε όλες τις γνωστές σωστές εξόδους ώστε να κάνουμε τη σύγκριση καθώς παράγονται τα αποτελέσματα από τον πολλαπλασιαστη, χρησιμοποιούμε έναν συμπιεστή διανυσμάτων ο οποίος εφαρμόζει τη λογική πράξη XOR. Έτσι μπορούμε να συγκρίνουμε την τελευταία τιμή εξόδου του συμπιεστή με την γνωστή και αναμενόμενη που έχουμε υπολογίζει προηγουμένως σε κάποιο σύστημα το οποίο δεν έχουμε αμφιβολία οτι μπορεί να σφάλει.

Οι τρείς μέθοδοι αυτοδοκιμής που εφαρμόσαμε είναι η LFSR, η Deterministic Counter και η ATPG. Οι μονάδες που παράγουν τα αντίστοιχα διανύσματα περιγράφονται παρακάτω και για κάθε μια παρατίθεται και η αντίστοιχη software υλοποίηση που μπορούμε επίσης να φορτώσουμε και να εκτελέσουμε στον επεξεργαστή μας για σύγκριση. Στο τέλος περιγράφονται και οι βοηθητικές για το σύστημα αυτοδοκιμής μονάδες όπως το control, ο συμπιεστής (MISR) και ο συγκριτής του τελικού αποτελέσματος.

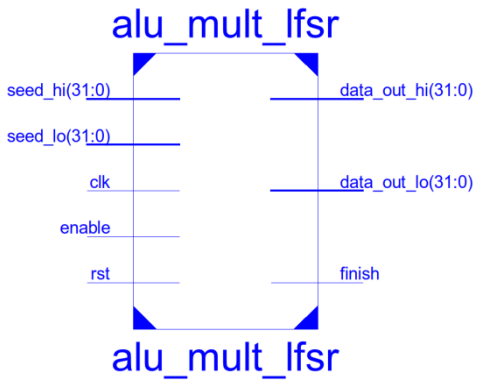
LFSR

Γενική περιγραφή

Η μονάδα αυτή παράγει ντερμινιστικά ψευδοτυχαίους αριθμούς των 64 bits και αρχικοποιείται από μια αρχική 64 bit τιμή που της δίνουμε σαν είσοδο. Μετά από ένα σημείο και μετά οι αριθμοί που παράγονται επαναλαμβάνονται οπότε έχουμε συγκρατήσει σε μια σταθερά τον τελευταίο ξεχωριστό αριθμό ώστε η διαδικασία να σταματήσει μόλις γίνει η παραγωγή αυτού του αριθμού.

Για την εκκίνηση της παραγωγής διανυσμάτων η μονάδα ελέγχου του πολλαπλασιαστή ενεργοποιεί το σήμα enable ενώ όταν ολοκληρωθεί η παραγωγή των διανυσμάτων η συγκεκριμένη (LFSR) μονάδα ενεργοποιεί το σήμα finish το οποίο και αναμένεται από τη μονάδα ελέγχου για να συνεχίσει με την επόμενη μέθοδο αυτοδοκιμής.

Block διάγραμμα



VHDL κώδικας

```
library ieee;
use ieee.std_logic_1164.all;

entity alu_mult_lfsr is
    port(   clk         : in  std_logic;
            rst         : in  std_logic;
            enable      : in  std_logic;
            finish      : out std_logic;
            seed_hi     : in  std_logic_vector(31 downto 0);
            seed_lo     : in  std_logic_vector(31 downto 0);
            data_out_hi : out std_logic_vector(31 downto 0);
            data_out_lo : out std_logic_vector(31 downto 0));
end alu_mult_lfsr;

architecture Behavioral of alu_mult_lfsr is

    constant last_vector    : std_logic_vector(63 downto 0) :=
X"5F670D48BECE1A91";
    signal lfsr_reg         : std_logic_vector(63 downto 0);

begin

    process (clk, rst, enable, seed_hi, seed_lo, lfsr_reg)
        variable lfsr_tap : std_logic;
    begin
        if(rst = '1') then
            lfsr_reg <= (others => '0');
            finish   <= '0';
        else
            if (clk'event and clk = '1') then
                if enable = '0' then
                    lfsr_reg(63 downto 32)  <= seed_hi;
                    lfsr_reg(31 downto 0)   <= seed_lo;
                    finish                  <= '0';
                elsif enable = '1' and lfsr_reg = last_vector then
                    lfsr_reg    <= (others => '0');
                    finish      <= '1';
                else
                    lfsr_tap    := lfsr_reg(0) xor lfsr_reg(1) xor lfsr_reg(3)
xor lfsr_reg(4);
                    lfsr_reg    <= lfsr_reg(62 downto 0) & lfsr_tap;
                    finish      <= '0';
                end if;
            end if;
        end if;
    end process;

    data_out_hi <= lfsr_reg(63 downto 32);
    data_out_lo <= lfsr_reg(31 downto 0);

end Behavioral;
```

Software Test

```
# ================================
# LFSR assembly program
# ================================
```

```
        .set noat

        .globl main                     # Call main by SPIM

        .text                           # Text section

main:   #lui     $10, 0xC018            # Correct result HI
        #ori     $10, $10, 0xCB25
        #lui     $11, 0x1750            # Correct result LO
        #ori     $11, $11, 0xF803

        lui     $10, 0x00000            # Correct result 32 bits (Edit this!)
        ori     $10, $10, 0x0000

        lui     $15, 0x0123             # LFSR Seed
        ori     $15, $15, 0x4567
        lui     $16, 0x89AB             # LFSR Seed
        ori     $16, $16, 0xCDEF

        addi    $12, $0, 0x00           # Last vector before repeat (Edit!)

        # Prepare registers for MISR
        lui     $22, 0x1800
        ori     $22, $22, 0x0002
        not     $23, $0
        addi    $6, $0, 0
        add     $9, $1, $0

        addi    $1, $0, 0               # i = 0

lfsr:   andi    $2, $9, 1
        andi    $3, $9, 2
        andi    $4, $9, 8
        andi    $5, $9, 16
        srl     $3, $3, 0x1
        srl     $4, $4, 0x3
        srl     $5, $5, 0x4
        xor     $2, $2, $3
        xor     $2, $2, $4
        xor     $2, $2, $5
        sll     $9, $9, 1
        or      $9, $9, $2
        mult    $9, $9
        mfhi    $3
        mflo    $4
        xor     $13, $3, $4

misr:   sll     $24, $23, 0x001f
        sra     $25, $24, 0x001f
        and     $25, $25, $22
        xor     $23, $23, $25
        srl     $23, $23, 0x0001
        addu    $23, $23, $24
        xor     $23, $23, $13

        addi    $1, $1, 1
        bne     $14, $12, lfsr
        slt     $13, $10, $23           # Check misr signature in $23 with
correct result

end:
```

Counter

Γενική περιγραφή

Η μονάδα αυτή όπως λέει και το όνομα της υλοποιεί έναν απλό μετρητή που αυξάνει από το 0 εως το 255. Οι αριθμοί που δίνονται σαν είσοδοι στον πολλαπλασιαστή είναι οι τιμές τα 4 MSBs και LSBs του μετρητή ενωμένα κατά 8 φορές όπως φαίνεται παρακάτω:

Είσοδος A: C(7:4)& C(7:4)& C(7:4)& C(7:4)& C(7:4)& C(7:4)& C(7:4)& C(7:4)
Είσοδος B: C(3:0)& C(3:0)& C(3:0)& C(3:0)& C(3:0)& C(3:0)& C(3:0)& C(3:0)

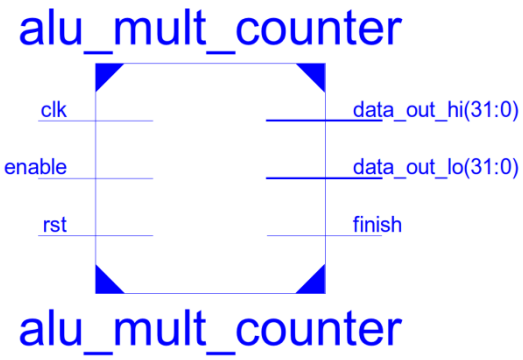
Για παράδειγμα αν ο μετρητής έχει την τιμή 0x45 τότε ο πολλαπλασιαστής θα πάρει για εισόδους:

Input HI = 0x44444444

Input LO = 0x55555555

Για την εκκίνηση της παραγωγής διανυσμάτων η μονάδα ελέγχου του πολλαπλασιαστή ενεργοποιεί το σήμα enable ενώ όταν ολοκληρωθεί η παραγωγή των διανυσμάτων η συγκεκριμένη (Counter) μονάδα ενεργοποιεί το σήμα finish το οποίο και αναμένεται από τη μονάδα ελέγχου για να συνεχίσει με την επόμενη μέθοδο αυτοδοκιμής.

Block διάγραμμα



VHDL κώδικας

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity alu_mult_counter is
    port (
        clk         : in  std_logic;
        rst         : in  std_logic;
        enable      : in  std_logic;
        finish      : out std_logic;
        data_out_hi : out std_logic_vector(31 downto 0);
        data_out_lo : out std_logic_vector(31 downto 0));
end entity alu_mult_counter;
```

```
architecture Behavioral of alu_mult_counter is

    signal counter      : std_logic_vector(7 downto 0);
    signal counter_hi   : std_logic_vector(3 downto 0);
    signal counter_lo   : std_logic_vector(3 downto 0);

begin

    process (clk, rst, enable)
    begin
        if(rst = '1') then
            counter <= (others => '0');
            finish  <= '0';
        else
            if (clk'event and clk='1') then
                if enable = '0' then
                    counter <= (others => '0');
                    finish  <= '0';
                elsif enable = '1' and counter = X"FF" then
                    counter <= (others => '0');
                    finish  <= '1';
                else
                    counter <= std_logic_vector(unsigned(counter) + 1);
                    finish  <= '0';
                end if;
            end if;
        end if;
    end process;

    counter_hi  <= counter(7 downto 4);
    counter_lo  <= counter(3 downto 0);

    data_out_hi <= counter_hi & counter_hi & counter_hi & counter_hi &
counter_hi & counter_hi & counter_hi & counter_hi;
    data_out_lo <= counter_lo & counter_lo & counter_lo & counter_lo &
counter_lo & counter_lo & counter_lo & counter_lo;

end architecture Behavioral;
```

Software Test

```
# ================================
# Counter SBST assembly program
# ================================

        .set noat

        .globl main                     # Call main by SPIM

        .text                           # Text section

main:   #lui     $10, 0xF4DA            # Correct result HI
        #ori     $10, $10, 0x9748
        #lui     $11, 0xF9DB            # Correct result LO
        #ori     $11, $11, 0xB48A

        lui     $10, 0x00000            # Correct result 32 bits (Edit this!)
        ori     $10, $10, 0x0000

        addi    $12, $0, 256            # Counter limit = 256
```

```
        # Prepare registers for MISR
        lui     $22, 0x1800
        ori     $22, $22, 0x0002
        not     $23, $0

        add     $1, $0, $0              # i = 0
        add     $2, $0, $0

count:  add     $2, $1, $0              # Set 0th byte
        sll     $2, $2, 8
        or      $2, $2, $1              # Set 1st byte
        sll     $2, $2, 8
        or      $2, $2, $1              # Set 2nd byte
        sll     $2, $2, 8
        or      $2, $2, $1              # Set 3rd byte
        mult    $2, $2
        mfhi    $3
        mflo    $4
        xor     $13, $3, $4

misr:   sll     $24, $23, 0x001f
        sra     $25, $24, 0x001f
        and     $25, $25, $22
        xor     $23, $23, $25
        srl     $23, $23, 0x0001
        addu    $23, $23, $24
        xor     $23, $23, $13

        addi    $1, $1, 1
        bne     $1, $12, count
        slt     $13, $10, $23           # Check misr signature in $23 with
correct result

end:
```

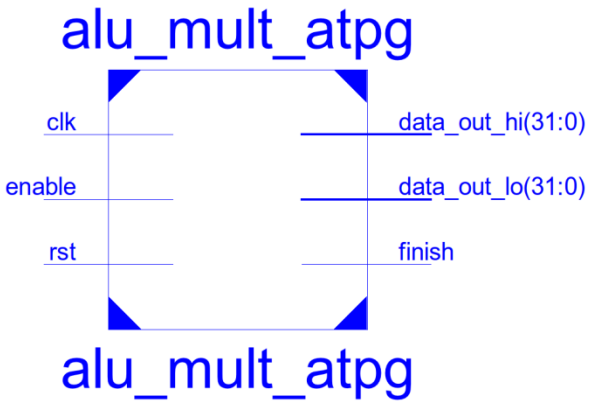
ATPG

Γενική περιγραφή

Η μονάδα αυτή περιλαμβάνει έναν μετρητή και δύο BRAMs με τα 64 bit διανύσματα (32 + 32) που δημιουργήσαμε χρησιμοποιώντας το εργαλείο δοκιμής της Synopsys. Για να μας παράξει το εργαλείο δοκιμής τα διανύσματα του δώσαμε το κύκλωμα του πολλαπλασιαστή σε μορφή πυλών (netlist) και αυτό το ανέλυσε ώστε να βρεί όλα τα πιθανά σφάλματα που μπορεί να προκύψουν. Ο αριθμός των διανυσμάτων που παρήγαγε το εργαλείο είναι 108.

Για την εκκίνηση της παραγωγής διανυσμάτων η μονάδα ελέγχου του πολλαπλασιαστή ενεργοποιεί το σήμα enable ενώ όταν ολοκληρωθεί η παραγωγή των διανυσμάτων η συγκεκριμένη (ATPG) μονάδα ενεργοποιεί το σήμα finish το οποίο και αναμένεται από τη μονάδα ελέγχου για να τερματίσει τη διαδικασία αυτοδοκιμής.

Block διάγραμμα



VHDL κώδικας

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity alu_mult_atpg is
    port (
        clk             : in  std_logic;
        rst             : in  std_logic;
        enable          : in  std_logic;
        finish          : out std_logic;
        data_out_hi     : out std_logic_vector(31 downto 0);
        data_out_lo     : out std_logic_vector(31 downto 0));
end entity alu_mult_atpg;

architecture Behavioral of alu_mult_atpg is

    component alu_mult_atpg_bram_hi
        port(clk         : in  std_logic;
             en          : in  std_logic;
             address     : in  std_logic_vector(6 downto 0);
             data_vector : out std_logic_vector(31 downto 0));
    end component alu_mult_atpg_bram_hi;

    component alu_mult_atpg_bram_lo
        port(clk         : in  std_logic;
             en          : in  std_logic;
             address     : in  std_logic_vector(6 downto 0);
             data_vector : out std_logic_vector(31 downto 0));
    end component alu_mult_atpg_bram_lo;

    signal address          : std_logic_vector(6 downto 0);
    constant last_address   : std_logic_vector(6 downto 0) := "1101100"; --
Total vectors = 108

begin

    process(clk, rst, enable)
    begin
        if(rst = '1') then
            address <= (others => '0');
            finish  <= '0';
```

```
        else
            if (clk'event and clk = '1') then
                if (enable = '0') then
                    address <= (others => '0');
                    finish  <= '0';
                elsif enable = '1' and address = last_address then
                    address <= (others => '0');
                    finish  <= '1';
                else
                    address <= std_logic_vector(unsigned(address) + 1);
                    finish  <= '0';
                end if;
            end if;
        end if;
    end process;

    BRAM_HI : alu_mult_atpg_bram_hi
        port map(clk         => clk,
                 en          => enable,
                 address     => address,
                 data_vector => data_out_hi);

    BRAM_LO : alu_mult_atpg_bram_lo
        port map(clk         => clk,
                 en          => enable,
                 address     => address,
                 data_vector => data_out_lo);


end architecture Behavioral;
```

Software Test

```
# ================================
# ATPG assembly program
# ================================

        .set noat

        .globl main                     # Call main by SPIM

        .text                           # Text section

main:   #lui     $10, 0xC921             # Correct result HI
        #ori     $10, $10, 0xB21F
        #lui     $11, 0x4172            # Correct result LO
        #ori     $11, $11, 0x1733

        lui     $10, 0x00000            # Correct result 32 bits (Edit this!)
        ori     $10, $10, 0x0000

        addi    $12, $0, 0x6C           # Vector limit = 108

        # Prepare registers for MISR
        lui     $22, 0x1800
        ori     $22, $22, 0x0002
        not     $23, $0
```

```
        add     $14, $0, $0             # Address = 0
        add     $1, $0, $0              # i = 0

        lw      $3, 0x000($1)           # First vector address (0)
        lw      $4, 0x1B0($1)           # Last vector address (108th vector X 4
= 432 = 0x1B0)

atpg:   mult    $3, $4
        mfhi    $3
        mflo    $4
        xor     $13, $3, $4

misr:   sll     $24, $23, 31
        sra     $25, $24, 31
        and     $25, $25, $22
        xor     $23, $23, $25
        srl     $23, $23, 0x0001
        addu    $23, $23, $24
        xor     $23, $23, $13

        addi    $14, $14, 4
        addi    $1, $1, 1
        bne     $14, $12, atpg
        slt     $13, $10, $23           # Check misr signature in $23 with
correct result

end:
```

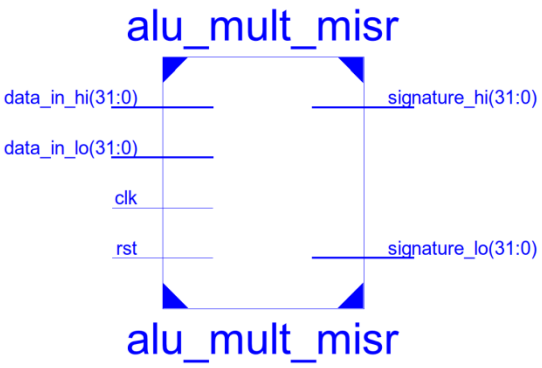
MISR + Comparator

Γενική περιγραφή

Οι μονάδες αυτές είναι υπεύθυνες για τη συμπίεση (compaction NOT compression) των αποτελεσμάτων του πολλαπλασιαστή και τη σύγκριση τους με τις γνωστές σωστές τιμές – υπογραφές. Χρησιμοποιούνται και στις τρείς παραπάνω μεθόδους αυτοδοκιμής και ο στόχος τους είναι να κάνουν την τελική επιλογή για το αν το κύκλωμα του πολλαπλασιαστή είναι ελλατωματικό.

Πιο συγκεκριμένα η μονάδα MISR δέχεται στη είσοδο της συνεχώς τις εξόδους του πολλαπλασιαστή και με ένα κύκλωμα που υλοποιεί ένα πολυώνυμο (όπως και στη μέθοδο LFSR) τις συμπιέζει παράγοντας συνεχώς μια υπογραφή η οποία συγκρατεί όλη την "ιστορία" των εξόδων μέχρι και εκείνο το σημείο. Συνεπώς αφού υπολογιστεί και το τελευταίο αποτέλεσμα από τον πολλαπλασιαστή υπολογίζεται η τελική υπογραφή η οποία και δίνεται στον συγκριτή ο οποίος με τη σειρά του τη συγκρίνει με τη γνωστή σωστή τιμή και εαν διαπιστώσει οποιαδήποτε αλλαγή σηκώνει το σήμα αποτυχίας που ειδοποιεί το χρήστη για το γεγονός αυτό.

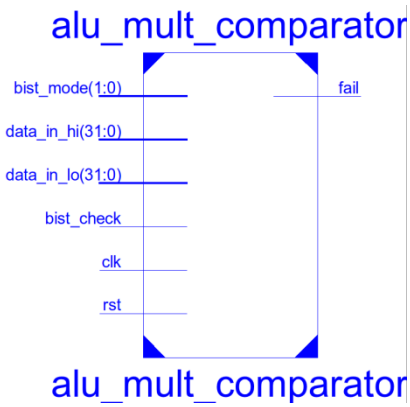
Block διάγραμμα



VHDL κώδικας

```
library ieee;
use ieee.std_logic_1164.all;

entity alu_mult_misr is
    port(   clk             : in  std_logic;
            rst             : in  std_logic;
            data_in_hi      : in  std_logic_vector(31 downto 0);
            data_in_lo      : in  std_logic_vector(31 downto 0);
            signature_hi    : out std_logic_vector(31 downto 0);
            signature_lo    : out std_logic_vector(31 downto 0));
end alu_mult_misr;

architecture Behavioral of alu_mult_misr is

    signal lfsr_reg : std_logic_vector(63 downto 0);
    signal data_in  : std_logic_vector(63 downto 0);

begin

    data_in <= data_in_hi & data_in_lo;

    process(clk, rst)
        variable lfsr_tap : std_logic;
    begin

    if(clk'event and clk = '1') then
        if rst = '1' then
            lfsr_reg    <= data_in;
        else
            lfsr_tap    := lfsr_reg(0) xor lfsr_reg(1) xor lfsr_reg(3) xor
lfsr_reg(4);
            lfsr_reg    <= (lfsr_reg(62 downto 0) & lfsr_tap) xor data_in;
        end if;
    end if;

    end process;

    signature_hi    <= lfsr_reg(63 downto 32);
    signature_lo    <= lfsr_reg(31 downto 0);

end Behavioral;
```

Block Διάγραμμα



VHDL κώδικας

```
library ieee;
use ieee.std_logic_1164.all;

entity alu_mult_comparator is
    port (
        clk         : in  std_logic;
        rst         : in  std_logic;
        bist_check  : in  std_logic;
        bist_mode   : in  std_logic_vector(1 downto 0);
        data_in_hi  : in  std_logic_vector(31 downto 0);
        data_in_lo  : in  std_logic_vector(31 downto 0);
        fail        : out std_logic);
end entity alu_mult_comparator;

architecture Behavioral of alu_mult_comparator is

    constant lfsr_hi_correct    : std_logic_vector(31 downto 0) :=
X"C018CB25";
    constant lfsr_lo_correct    : std_logic_vector(31 downto 0) :=
X"1750F803";
    constant counter_hi_correct : std_logic_vector(31 downto 0) :=
X"F4DA9748";
    constant counter_lo_correct : std_logic_vector(31 downto 0) :=
X"F9DBB48A";
    constant atpg_hi_correct    : std_logic_vector(31 downto 0) :=
X"C921B21F";
    constant atpg_lo_correct    : std_logic_vector(31 downto 0) :=
X"41721733";

begin

    process (clk, rst, bist_check, bist_mode, data_in_hi, data_in_lo)
    begin
        if(rst = '1') then
            fail <= '0';
        else
            if (clk'event and clk='1') then
                if bist_check = '1' then
                    if(bist_mode = "01" and (data_in_hi /= lfsr_hi_correct
or data_in_lo /= lfsr_lo_correct)) then
                        fail <= '1';
                    elsif(bist_mode = "10" and (data_in_hi /=
counter_hi_correct or data_in_lo /= counter_lo_correct)) then
                        fail <= '1';
```

```
                    elsif(bist_mode = "11" and (data_in_hi /=
atpg_hi_correct or data_in_lo /= atpg_lo_correct)) then
                        fail <= '1';
                    else
                        fail <= '0';
                    end if;
                end if;
            end if;
        end if;
    end process;

end architecture Behavioral;
```

Control

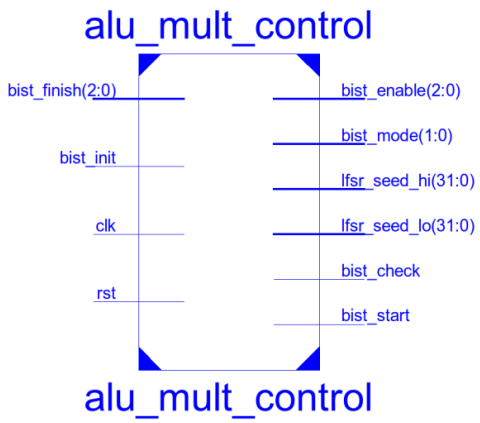
Γενική περιγραφή

Η μονάδα ελέγχου είναι υπεύθυνη για την ενορχήστρωση όλων των υπομονάδων που απαρτίζουν την ευρύτερη μονάδα του πολλαπλασιαστή. Η μονάδα αυτή είναι καθαρά συνδυαστική και περιλαμβάνει μια μηχανή πεπερασμένων καταστάσεων (FSM) τύπου Moore υλοποιημένη με 3 process. Στην πραγματικότητα έχουμε αφαιρέσει το τρίτο process για τον καθορισμό των εξόδων ανάλογα την κατάσταση και το έχουμε αντικαταστήσει με συνδυαστική λογική χωρίς process. Όλα αυτά είναι σύμφωνα με τα σωστά πρότυπα συγγραφής VHDL και συνιστούνται κατά τη συγγραφή μιας FSM τύπου Moore (ή και Mealy).

Κατά την κανονική λειτουργία του επεξεργαστή η μονάδα αυτή παραμένει στην αρχική της κατάσταση. Μόλις ο επεξεργαστής δεχθεί την εντολή TEST η κεντρική μονάδα ελέγχου ενεργοποιεί το σήμα TestMult το οποίο και καταλήγει στην είσοδο της συγκεκριμένης μονάδας ελέγχου (bist_init). Αυτό οδηγεί την FSM στην επόμενη κατάσταση από την οποία και ξεκινάει ένα loop για την εκτέλεση και των τριών παραπάνω μεθόδων αυτοδοκιμής στέλνοντας και λαμβάνοντας τα απαραίτητα για το συγχρονισμό σήματα. Η συγκεκριμένη FSM υποστηρίζει και τους δύο τύπους πολλαπλασιαστή που έχουμε υλοποιήσει (pipelined, κανονικός) ελέγχοντας την generic παράμετρο mult_pipe και υλοποιώντας καταστάσεις αναμονής του αποτελέσματος του πολλαπλασιαστή όπου αυτό απαιτείται (πριν το πρώτο αποτέλεσμα και πριν το τελευταίο) ώστε να επιτευχθεί ο σωστός συγχρονισμός.

Η μονάδα ελέγχου του πολλαπλασιαστή επιστρέφει τον έλεγχο στην κεντρική μονάδα μόλις ολοκληρωθούν και οι τρείς μέθοδοι αυτοδοκιμής ώστε να μπορέσει ο επεξεργαστής να φορτώσει και να εκτελέσει και πάλι κανονικά τις εντολές από τη μνήμη εντολών.

Block διάγραμμα



VHDL κώδικας

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity alu_mult_control is
    port (
        clk             : in  std_logic;
        rst             : in  std_logic;
        bist_init       : in  std_logic;
        bist_finish     : in  std_logic_vector(2 downto 0);
        bist_start      : out std_logic;
        bist_check      : out std_logic;
        bist_mode       : out std_logic_vector(1 downto 0);
        bist_enable     : out std_logic_vector(2 downto 0);
        lfsr_seed_hi    : out std_logic_vector(31 downto 0);
        lfsr_seed_lo    : out std_logic_vector(31 downto 0));
end entity alu_mult_control;

architecture Behavioral of alu_mult_control is

    type state_type is (s0, s1, s2, s3a, s3b, s4, s5, s6, s7);
    signal state, next_state : state_type;

    constant seed_hi : std_logic_vector(31 downto 0)    := X"01234567"; --
Insert LFSR HI seed here
    constant seed_lo : std_logic_vector(31 downto 0)    := X"89ABCDEF"; --
Insert LFSR LO seed here

    constant before_cycles_normal   : std_logic_vector(2 downto 0) := "100";
    constant before_cycles_atpg     : std_logic_vector(2 downto 0) := "101";
    constant after_cycles           : std_logic_vector(2 downto 0) := "011";

    signal bist_method      : std_logic_vector(1 downto 0);
    signal bist_active      : std_logic_vector(2 downto 0);
    signal bist_done        : std_logic;
    signal before_counter   : std_logic_vector(2 downto 0);
    signal after_counter    : std_logic_vector(2 downto 0);


begin

    -- ================================
    -- Registered process, updates state
    -- ================================

    registered: process (clk, rst)
    begin

        if(rst = '1') then

            state           <= s0;
            bist_done       <= '0';
            before_counter  <= (others => '0');
            after_counter   <= (others => '0');
            bist_method     <= (others => '0');
            bist_active     <= (others => '0');

        elsif(clk'event and clk = '1') then

            case state is
```

```
                when s1     =>      if(bist_method = "11") then
                                        bist_method <= (others => '0');
                                    else
                                        bist_method <=
std_logic_vector(unsigned(bist_method) + 1);
                                    end if;

                when s2     =>      case bist_method is
                                        when "00" => bist_active <= "000";
                                        when "01" => bist_active <= "001";
                                        when "10" => bist_active <= "010";
                                        when "11" => bist_active <= "100";
                                        when others => bist_active <= "000";
                                    end case;

                when s3a    =>      if(before_counter = before_cycles_normal)
then
                                        before_counter <= (others => '0');
                                    else
                                        before_counter <=
std_logic_vector(unsigned(before_counter) + 1);
                                    end if;

                when s3b    =>      if(before_counter = before_cycles_atpg) then
                                        before_counter <= (others => '0');
                                    else
                                        before_counter <=
std_logic_vector(unsigned(before_counter) + 1);
                                    end if;

                when s6     =>      if(after_counter = after_cycles) then
                                        after_counter <= (others => '0');
                                    else
                                        after_counter <=
std_logic_vector(unsigned(after_counter) + 1);
                                    end if;

                when s7   =>        if(bist_method = "11") then
                                        bist_done <= '1';
                                    end if;

                when others => null;

            end case;

            bist_mode <= bist_method;

            state <= next_state;

        end if;

    end process registered;

    -- ================================================================
    -- Combinational process, changes state based on current state and input
    -- ================================================================

    combinational : process (state, bist_init, bist_finish, bist_method,
bist_done, before_counter, after_counter)
    begin

        next_state <= state;
```

```
        case state is

                -- ==============
                -- COMMAND STATES
                -- ==============

                -- Normal operation
                when s0 =>      if bist_init = '1' and bist_done = '0' then
                                    next_state <= s1;
                                else
                                    next_state <= s0;
                                end if;

                -- BIST

                -- Main BIST loop for all 3 methods (LFSR, Counter, ATPG)
                when s1 =>      if(bist_done = '1') then
                                    next_state <= s0;
                                else
                                    next_state <= s2;
                                end if;

                -- Init BIST method
                when s2 =>      if(bist_method /= "11") then
                                    next_state <= s3a;
                                else
                                    next_state <= s3b;
                                end if;

                -- Wait for the first multiplier result
                when s3a =>      if(before_counter = before_cycles_normal) then
                                    next_state <= s4;
                                else
                                    next_state <= s3a;
                                end if;

                -- Wait for the first multiplier result
                when s3b =>      if(before_counter = before_cycles_atpg) then
                                    next_state <= s4;
                                else
                                    next_state <= s3b;
                                end if;

                -- Start BIST method
                when s4 =>      next_state <= s5;

                -- BIST method loop
                when s5 =>      if bist_finish /= "000" then
                                    next_state <= s6;
                                else
                                    next_state <= s5;
                                end if;

                -- Wait for the last MISR signature that captured up to the last
multiplier result
                when s6 =>      if(after_counter = after_cycles) then
                                    next_state <= s7;
                                else
                                    next_state <= s6;
                                end if;

                -- Check signature and go to next method
                when s7 =>      next_state <= s1;
```

```
                -- Not needed because all states are covered
                --when others =>  next_state <= s0;

        end case;

    end process combinational;

    -- ============
    -- Output Logic
    -- ============

    bist_start  <= '1' when state = s4 else '0';
    bist_check  <= '1' when state = s7 else '0';
    bist_enable <= "000" when state = s0 else bist_active;

    lfsr_seed_hi    <= seed_hi;
    lfsr_seed_lo    <= seed_lo;

end architecture Behavioral;
```

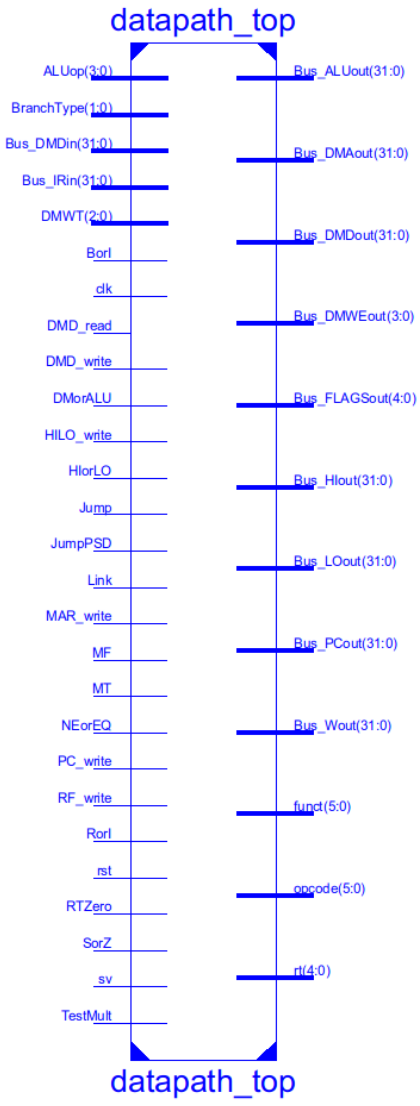
2.3 Τεχνική περιγραφή της διόδου δεδομένων (datapath)

Γενική περιγραφή

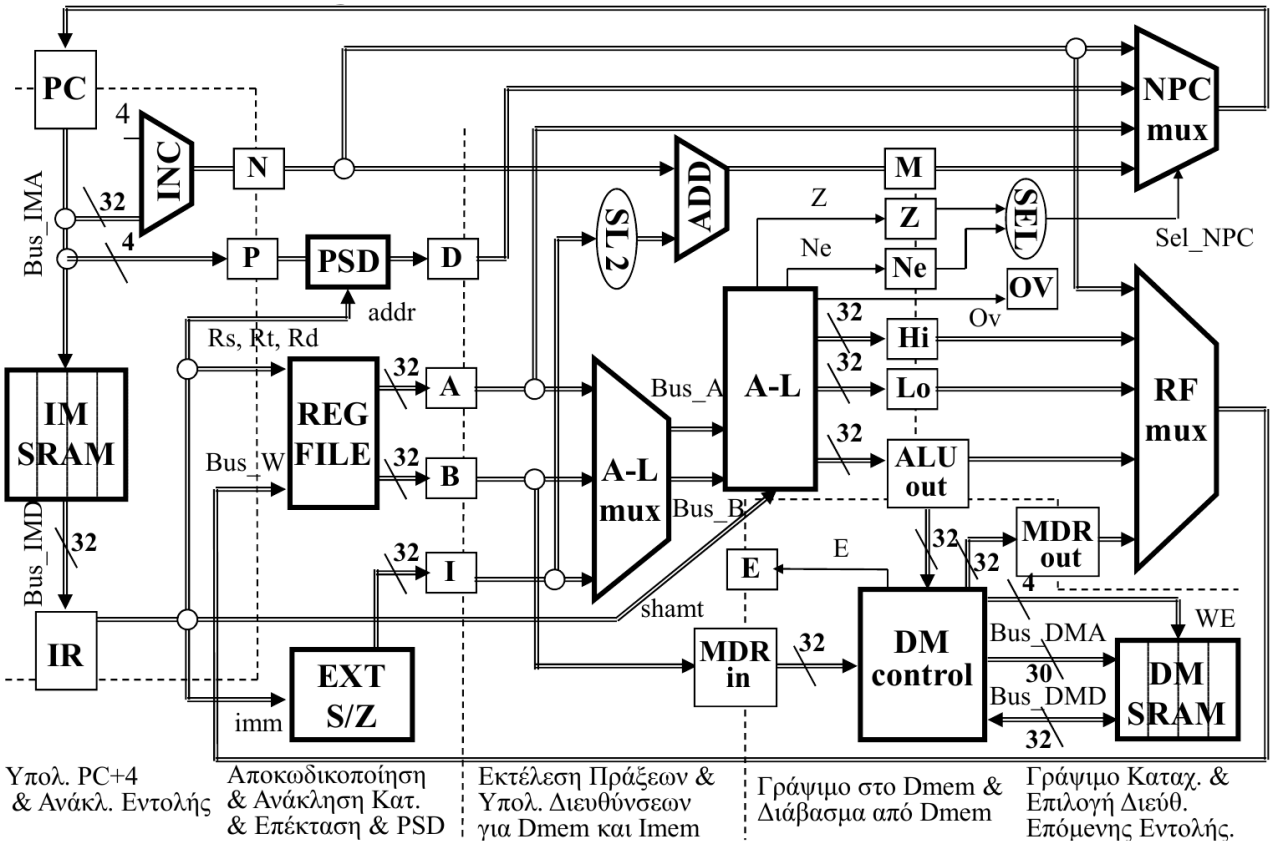
Η μονάδα της διόδου δεδομένων περιλαμβάνει όλες τις μονάδες που αναλύσαμε στις προηγούμενες ενότητες. Η διασύνδεση τους γίνεται με περιγραφή δομής και ακολουθεί κατά ένα μεγάλο μέρος τη διασύνδεση που προτείνεται στις σημειώσεις του μαθήματος. Κατά την εκτέλεση της κάθε εντολής είναι ενεργές μόνο συγκεκριμένες μονάδες ανάλογα με τα σήματα ελέγχου που δέχεται από τη μονάδα ελέγχου. Η δίοδος δεδομένων επικοινωνεί εξωτερικά με τις δύο μνήμες, εντολών και δεδομένων και με τη μονάδα ελέγχου.

Για τις περισσότερες εντολές ένας κύκλος εκτέλεσης (4 ή περισσοτέρων κύκλων ρολογιού) της διόδου δεδομένων ξεκινάει οταν προσκομιστεί μια νέα εντολή από τη μνήμη εντολών και τερματίζει οταν εγγραφή το αποτέλεσμα της εντολής σε κάποιον καταχωρητή ή στη μνήμη δεδομένων.
Φυσικά υπάρχουν και άλλου είδους εντολές όπως οι Branch και Jump οι οποίες αλλάζουν τη ροή του προγράμματος και τερματίζουν σε λιγότερους κύκλους ρολογιού (π.χ 3).

Block διάγραμμα



Σχηματικό



VHDL κώδικας

```
library ieee;
use ieee.std_logic_1164.all;

entity datapath_top is
    generic(mult_pipe       : boolean := true);
    port(   clk             : in  std_logic;
            rst             : in  std_logic;
            PC_write        : in  std_logic;
            RF_write        : in  std_logic;
            MAR_write       : in  std_logic;
            DMD_read        : in  std_logic;
            DMD_write       : in  std_logic;
            HILO_write      : in  std_logic;
            RorI            : in  std_logic;
            SorZ            : in  std_logic;
            BorI            : in  std_logic;
            sv              : in  std_logic;
            MF              : in  std_logic;
            MT              : in  std_logic;
            HIorLO          : in  std_logic;
            Jump            : in  std_logic;
            JumpPSD         : in  std_logic;
            BranchType      : in  std_logic_vector(1 downto 0);
            NEorEQ          : in  std_logic;
            RTZero          : in  std_logic;
            Link            : in  std_logic;
            DMorALU         : in  std_logic;
            DMWT            : in  std_logic_vector(2 downto 0);
            TestMult        : in  std_logic;
```

```
            ALUop           : in  std_logic_vector(3 downto 0);
            Bus_IRin        : in  std_logic_vector(31 downto 0);
            Bus_DMDin       : in std_logic_vector(31 downto 0);
            opcode          : out std_logic_vector(5 downto 0);
            funct           : out std_logic_vector(5 downto 0);
            rt              : out std_logic_vector(4 downto 0);
            Bus_FLAGSout    : out std_logic_vector(4 downto 0);
            Bus_PCout       : out std_logic_vector(31 downto 0);
            Bus_ALUout      : out std_logic_vector(31 downto 0);
            Bus_HIout       : out std_logic_vector(31 downto 0);
            Bus_LOout       : out std_logic_vector(31 downto 0);
            Bus_Wout        : out std_logic_vector(31 downto 0);
            Bus_DMWEout     : out std_logic_vector(3 downto 0);
            Bus_DMAout      : out std_logic_vector(31 downto 0);
            Bus_DMDout      : out std_logic_vector(31 downto 0));
end datapath_top;

architecture Structural of datapath_top is

    component reg_we is
        generic(    W       : integer := 32);
        port(   clk         : in  std_logic;
                rst         : in  std_logic;
                we          : in  std_logic;
                data_in     : in  std_logic_vector(W - 1 downto 0);
                data_out    : out std_logic_vector(W - 1 downto 0));
    end component;

    component reg is
        generic ( W         : integer := 32);
        port(   clk         : in  std_logic;
                rst         : in  std_logic;
                data_in     : in  std_logic_vector(W - 1 downto 0);
                data_out    : out std_logic_vector(W - 1 downto 0));
    end component;

    component rf_32x32 is
        port(   clk         : in  std_logic;
                rst         : in  std_logic;
                RegWrite    : in  std_logic;
                RegImmNot   : in  std_logic;
                RTZero      : in  std_logic;
                rs          : in  std_logic_vector(4 downto 0);
                rt          : in  std_logic_vector(4 downto 0);
                rd          : in  std_logic_vector(4 downto 0);
                dataW_in    : in  std_logic_vector(31 downto 0);
                dataA_out   : out std_logic_vector(31 downto 0);
                dataB_out   : out std_logic_vector(31 downto 0));
    end component;

    component rf_mux is
        port(   data_alu_in : in  std_logic_vector(31 downto 0);
                data_dm_in  : in  std_logic_vector(31 downto 0);
                data_npc_in : in  std_logic_vector(31 downto 0);
                data_mlo_in : in  std_logic_vector(31 downto 0);
                data_mhi_in : in  std_logic_vector(31 downto 0);
                Link        : in  std_logic;
                DMorALU     : in  std_logic;
                MF          : in  std_logic;
                HIorLO      : in  std_logic;
                data_out    : out std_logic_vector(31 downto 0));
    end component;
```

```
    component extend_immediate is
        port(   data_in      : in  std_logic_vector(15 downto 0);
                SorZ         : in  std_logic;
                data_out     : out std_logic_vector(31 downto 0));
    end component;

    component npc_adder is
        port(   dataA_in     : in  std_logic_vector(31 downto 0);
                dataB_in     : in  std_logic_vector(31 downto 0);
                data_out     : out std_logic_vector(31 downto 0));
    end component;

    component npc_inc is
        port(   data_in      : in  std_logic_vector(31 downto 0);
                data_out     : out std_logic_vector(31 downto 0));
    end component;

    component npc_sel is
        port(   Jump         : in  std_logic;
                JumpPSD      : in  std_logic;
                BranchType   : in  std_logic_vector(1 downto 0);
                NEorEQ       : in  std_logic;
                Zero         : in  std_logic;
                Negative     : in  std_logic;
                JumpSelect   : out std_logic_vector(1 downto 0));
    end component;

    component npc_mux is
        port(   data_npc_in  : in  std_logic_vector(31 downto 0);
                data_imm_in  : in  std_logic_vector(31 downto 0);
                data_reg_in  : in  std_logic_vector(31 downto 0);
                data_psd_in  : in  std_logic_vector(31 downto 0);
                JumpSelect   : in  std_logic_vector(1 downto 0);
                data_out     : out std_logic_vector(31 downto 0));
    end component;

    component npc_psd is
        port(   dataP_in     : in  std_logic_vector(3 downto 0);
                dataA_in     : in  std_logic_vector(25 downto 0);
                data_out     : out std_logic_vector(31 downto 0));
    end component;

    component npc_sl2 is
        port(   data_in      : in  std_logic_vector(31 downto 0);
                data_out     : out std_logic_vector(31 downto 0));
    end component;

    component alu_top is
        generic(mult_pipe    : boolean := true);
        port(   clk          : in  std_logic;
                rst          : in  std_logic;
                sv           : in  std_logic;
                TestMult     : in  std_logic;
                MT           : in  std_logic;
                HIorLO       : in  std_logic;
                ALUop        : in  std_logic_vector(3 downto 0);
                shamt        : in  std_logic_vector(4 downto 0);
                Bus_A        : in  std_logic_vector(31 downto 0);
                Bus_B        : in  std_logic_vector(31 downto 0);
                Zero         : out std_logic;
                ov           : out std_logic;
                bist_fail    : out std_logic;
                Bus_S        : out std_logic_vector(31 downto 0);
```

```
            Bus_mult_HI : out std_logic_vector(31 downto 0);
            Bus_mult_LO : out std_logic_vector(31 downto 0));
end component;

component alu_mux is
    port(   clk             : in std_logic;
            rst             : in std_logic;
            data_regA_in    : in  std_logic_vector(31 downto 0);
            data_regB_in    : in  std_logic_vector(31 downto 0);
            data_imm_in     : in  std_logic_vector(31 downto 0);
            AluSelect       : in  std_logic;
            dataA_out       : out std_logic_vector(31 downto 0);
            dataB_out       : out std_logic_vector(31 downto 0));
end component;

component dm_control is
    port(   data_mdr_in     : in  std_logic_vector(31 downto 0);
            data_mar_in     : in  std_logic_vector(31 downto 0);
            data_dmd_in     : in  std_logic_vector(31 downto 0);
            DMWT            : in  std_logic_vector(2 downto 0);
            DMD_we          : in  std_logic;
            Error           : out std_logic_vector(0 downto 0);
            data_mdr_out    : out std_logic_vector(31 downto 0);
            data_dma_out    : out std_logic_vector(31 downto 0);
            data_we_out     : out std_logic_vector(3 downto 0);
            data_dmd_out    : out std_logic_vector(31 downto 0));
end component;

signal Bus_PC       : std_logic_vector(31 downto 0);
signal Bus_INC      : std_logic_vector(31 downto 0);
signal Bus_NPC      : std_logic_vector(31 downto 0);
signal Bus_P        : std_logic_vector(3 downto 0);
signal Bus_PSD      : std_logic_vector(31 downto 0);
signal Bus_D        : std_logic_vector(31 downto 0);
signal Bus_SL2      : std_logic_vector(31 downto 0);
signal Bus_ADD      : std_logic_vector(31 downto 0);
signal Bus_M        : std_logic_vector(31 downto 0);
signal Bus_NPCSEL   : std_logic_vector(1 downto 0);
signal Bus_NPCMUX   : std_logic_vector(31 downto 0);

signal Bus_W        : std_logic_vector(31 downto 0);
signal Bus_RA       : std_logic_vector(31 downto 0);
signal Bus_RB       : std_logic_vector(31 downto 0);
signal Bus_A        : std_logic_vector(31 downto 0);
signal Bus_B        : std_logic_vector(31 downto 0);
signal Bus_I        : std_logic_vector(31 downto 0);
signal Bus_EXT      : std_logic_vector(31 downto 0);

signal Bus_ALUMUXA  : std_logic_vector(31 downto 0);
signal Bus_ALUMUXB  : std_logic_vector(31 downto 0);
signal Bus_ALU      : std_logic_vector(31 downto 0);
signal Bus_MULTHI   : std_logic_vector(31 downto 0);
signal Bus_MULTLO   : std_logic_vector(31 downto 0);
signal Bus_ALUFLAGS : std_logic_vector(3 downto 0);
signal Bus_FLAGS    : std_logic_vector(3 downto 0);
signal Zero         : std_logic;
signal Overflow     : std_logic;
signal bist_fail    : std_logic;
signal HI_we        : std_logic;
signal LO_we        : std_logic;

signal Bus_ALUO     : std_logic_vector(31 downto 0);
signal Bus_HI       : std_logic_vector(31 downto 0);
```

```
    signal Bus_LO       : std_logic_vector(31 downto 0);

    signal Error        : std_logic_vector(0 downto 0);
    signal FlagE        : std_logic_vector(0 downto 0);
    signal Bus_MDRI     : std_logic_vector(31 downto 0);
    signal Bus_MDRO     : std_logic_vector(31 downto 0);
    signal Bus_MAR      : std_logic_vector(31 downto 0);
    signal Bus_SHAMT    : std_logic_vector(4 downto 0);

begin

    opcode          <= Bus_IRin(31 downto 26);
    funct           <= Bus_IRin(5 downto 0);
    rt              <= BUS_IRin(20 downto 16);
    Bus_FLAGSout    <= FlagE(0) & Bus_FLAGS;
    Bus_PCout       <= Bus_PC;
    Bus_ALUout      <= Bus_ALU;
    Bus_HIout       <= Bus_HI;
    Bus_LOout       <= Bus_LO;
    Bus_Wout        <= Bus_W;
    Bus_ALUFLAGS    <= bist_fail & Overflow & Bus_ALU(31) & Zero;

    HI_we           <= HILO_write or (HILO_write and MT and HIorLO);
    LO_we           <= HILO_write or (HILO_write and MT and (not HIorLO));

    PC : reg_we
    port map(   clk         => clk,
                rst         => rst,
                we          => PC_write,
                data_in     => Bus_NPCMUX,
                data_out    => Bus_PC);

    NPC : reg
    port map(   clk         => clk,
                rst         => rst,
                data_in     => Bus_INC,
                data_out    => Bus_NPC);

    P : reg
    generic map(    W       => 4)
    port map(   clk         => clk,
                rst         => rst,
                data_in     => Bus_PC(31 downto 28),
                data_out    => Bus_P);

    D : reg
    port map(   clk         => clk,
                rst         => rst,
                data_in     => Bus_PSD,
                data_out    => Bus_D);

    A : reg
    port map(   clk         => clk,
                rst         => rst,
                data_in     => Bus_RA,
                data_out    => Bus_A);

    B : reg
    port map(   clk         => clk,
                rst         => rst,
                data_in     => Bus_RB,
                data_out    => Bus_B);
```

```
    I : reg
    port map(   clk         => clk,
                rst         => rst,
                data_in     => Bus_EXT,
                data_out    => Bus_I);

    M : reg
    port map(   clk         => clk,
                rst         => rst,
                data_in     => Bus_ADD,
                data_out    => Bus_M);

    S : reg
    generic map( W          => 5)
    port map(   clk         => clk,
                rst         => rst,
                data_in     => Bus_IRin(10 downto 6),
                data_out    => Bus_SHAMT);

    ALUOUT : reg
    port map(   clk         => clk,
                rst         => rst,
                data_in     => Bus_ALU,
                data_out    => Bus_ALUO);

    HI : reg_we
    port map(   clk         => clk,
                rst         => rst,
                we          => HI_we,
                data_in     => Bus_MULTHI,
                data_out    => Bus_HI);

    LO : reg_we
    port map(   clk         => clk,
                rst         => rst,
                we          => LO_we,
                data_in     => Bus_MULTLO,
                data_out    => Bus_LO);

    FLAGS : reg
    generic map(    W       => 4)
    port map(   clk         => clk,
                rst         => rst,
                data_in     => Bus_ALUFLAGS,
                data_out    => Bus_FLAGS);

    MDRI : reg
    port map(   clk         => clk,
                rst         => rst,
                data_in     => Bus_B,
                data_out    => Bus_MDRI);

    MAR : reg_we
    port map(   clk         => clk,
                rst         => rst,
                we          => MAR_write,
                data_in     => Bus_ALU,
                data_out    => Bus_MAR);

    ERR : reg
    generic map( W          => 1)
    port map(   clk         => clk,
                rst         => rst,
```

```
                data_in     => Error,
                data_out    => FlagE);

    RF : rf_32x32
    port map(   clk         => clk,
                rst         => rst,
                RegWrite    => RF_write,
                RegImmNot   => RorI,
                RTZero      => RTZero,
                rs          => Bus_IRin(25 downto 21),
                rt          => Bus_IRin(20 downto 16),
                rd          => Bus_IRin(15 downto 11),
                dataW_in    => Bus_W,
                dataA_out   => Bus_RA,
                dataB_out   => Bus_RB);

    RFMUX : rf_mux
    port map(   data_alu_in => Bus_ALUO,
                data_dm_in  => Bus_MDRO,
                data_npc_in => Bus_NPC,
                data_mlo_in => Bus_LO,
                data_mhi_in => Bus_HI,
                Link        => Link,
                DMorALU     => DMorALU,
                MF          => MF,
                HIorLO      => HIorLO,
                data_out    => Bus_W);

    EXTIMM : extend_immediate
    port map(   data_in     => Bus_IRin(15 downto 0),
                SorZ        => SorZ,
                data_out    => Bus_EXT);

    NPCADD : npc_adder
    port map(   dataA_in    => Bus_NPC,
                dataB_in    => Bus_SL2,
                data_out    => Bus_ADD);

    NPCINC : npc_inc
    port map(   data_in     => Bus_PC,
                data_out    => Bus_INC);

    NPCSEL : npc_sel
    port map(   Jump        => Jump,
                JumpPSD     => JumpPSD,
                BranchType  => BranchType,
                NEorEQ      => NEorEQ,
                Zero        => Bus_FLAGS(0),
                Negative    => Bus_FLAGS(1),
                JumpSelect  => Bus_NPCSEL);

    NPCMUX : npc_mux
    port map(   data_npc_in => Bus_NPC,
                data_imm_in => Bus_M,
                data_reg_in => Bus_A,
                data_psd_in => Bus_D,
                JumpSelect  => Bus_NPCSEL,
                data_out    => Bus_NPCMUX);

    NPCPSD : npc_psd
    port map(   dataP_in    => Bus_P,
                dataA_in    => Bus_IRin(25 downto 0),
                data_out    => Bus_PSD);
```

```
    NPCSL2 : npc_sl2
    port map(   data_in     => Bus_I,
                data_out    => Bus_SL2);

    ALU : alu_top
    generic map(mult_pipe   => mult_pipe)
    port map(   clk         => clk,
                rst         => rst,
                sv          => sv,
                TestMult    => TestMult,
                MT          => MT,
                HIorLO      => HIorLO,
                ALUop       => ALUop,
                shamt       => Bus_SHAMT,
                Bus_A       => Bus_ALUMUXA,
                Bus_B       => Bus_ALUMUXB,
                Zero        => Zero,
                ov          => Overflow,
                bist_fail   => bist_fail,
                Bus_S       => Bus_ALU,
                Bus_mult_HI => Bus_MULTHI,
                Bus_mult_LO => Bus_MULTLO);

    ALUMUX : alu_mux
    port map(   clk             => clk,
                rst             => rst,
                data_regA_in    => Bus_A,
                data_regB_in    => Bus_B,
                data_imm_in     => Bus_I,
                AluSelect       => BorI,
                dataA_out       => Bus_ALUMUXA,
                dataB_out       => Bus_ALUMUXB);

    DMCONTROL : dm_control
    port map(   data_mdr_in     => Bus_MDRI,
                data_mar_in     => Bus_MAR,
                data_dmd_in     => Bus_DMDin,
                DMWT            => DMWT,
                DMD_we          => DMD_write,
                Error           => Error,
                data_mdr_out    => Bus_MDRO,
                data_dma_out    => Bus_DMAout,
                data_we_out     => Bus_DMWEout,
                data_dmd_out    => Bus_DMDout);

end Structural;
```

2.4 Τεχνική περιγραφή της μονάδας ελέγχου (control unit)

Γενική περιγραφή

Η μονάδα ελέγχου είναι υπεύθυνη για την ενορχήστρωση όλων των υπομονάδων που απαρτίζουν το σύνολο του επεξεργαστή. Χωρίζεται σε δύο λογικά μέρη, ένα (κυρίως) συνδυαστικό και ένα ακολουθιακό. Το πρώτο είναι αυτό που δημιουργεί τα σήματα ελέγχου που μένουν σταθερά σε όλη τη διάρκεια της κάθε εντολής ανεξαρτήτως των κύκλων που διαρκεί αυτή, ενώ το δεύτερο δημιουργεί σήματα ελέγχου που εξαρτώνται από τη φάση εκτέλεσης στην οποία βρίσκεται η εντολή.

Συνδυαστική μονάδα

Γενική περιγραφή

Η μονάδα αυτή είναι κυρίως συνδυαστική και για κάθε μια εντολή, με βάση τα πεδία opcode και func, θέτει τις σωστές τιμές για ένα σύνολο από σήματα ελέγχου τα περισσότερα από τα οποία τα είδαμε στις προηγούμενες ενότητες.

Λέμε οτι είναι κυρίως συνδυαστική γιατι επιλέξαμε να τοποθετήσουμε καταχωρητές στην έξοδο όλων των σημάτων πλήν 3 για τα οποία δεν ήταν επιτρεπτό. Η τοποθέτηση καταχωρητών στις εξόδους ήταν απαραίτητη για τη μείωση των περισσοτέρων μονοπατιών καθυστέρησης της διόδου δεδομένων καθώς σχεδόν όλες οι μονάδες δέχονται σήματα από τη μονάδα ελέγχου. Τα μόνα σήματα για το οποία δεν ήταν επιτρεπτό να γίνει κάτι τέτοιο είναι αυτά τα οποία καταλήγουν στο αρχείο καταχωρητών (SorZ, RoRI, RTZero), το οποίο χρειάζεται τη τιμή τους στον δεύτερο κύκλο εκτέλεσης (ID) και όχι στον 3ο όπου εκεί σταθεροποιούνται τα σήματα ελέγχου στα οποία τοποθετούμε καταχωρητές εξόδου.

Ακολουθεί ο πίνακας όλων των σημάτων ελέγχου που παράγει η συνδυαστική μονάδα ελέγχου.

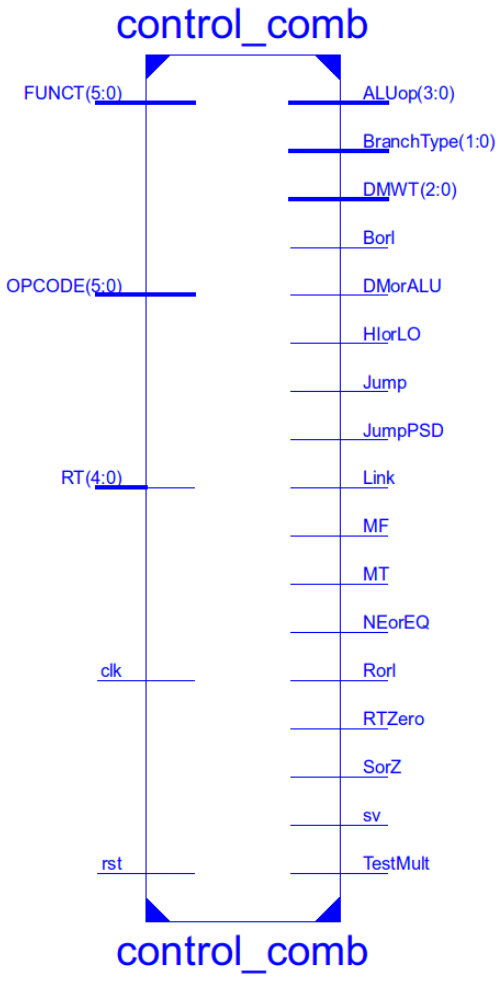
Πίνακας τιμών

Instruction	SorZ	Borl	ALUop	sv	MF	MT	HiorLO	DmorALU	DMWT	Link	Rorl	BranchType	NeorEQ	RTZero	Jump	JumpPSD	TestMult
LW	1	0	1001		0			1	100	0	0	0			0		0
LH	1	0	1001		0			1	011	0	0	0			0		0
LHU	1	0	1001		0			1	010	0	0	0			0		0
LB	1	0	1001		0			1	001	0	0	0			0		0
LBU	1	0	1001		0			1	000	0	0	0			0		0
SW	1	0	1001						100			0			0		0
SH	1	0	1001						011			0			0		0
SB	1	0	1001						001			0			0		0

ADDI	1	0	1000		0			0		0	0	0			0		0
ADDIU	1	0	1001		0			0		0	0	0			0		0
ANDI	0	0	1100		0			0		0	0	0			0		0
ORI	0	0	1101		0			0		0	0	0			0		0
XORI	0	0	1110		0			0		0	0	0			0		0
ADD		1	1000		0			0		0	1	0			0		0
ADDU		1	1001		0			0		0	1	0			0		0
SUB		1	1010		0			0		0	1	0			0		0
SUBU		1	1011		0			0		0	1	0			0		0
MULT		1								0		0			0		0
AND		1	1100		0			0		0	1	0			0		0
OR		1	1101		0			0		0	1	0			0		0
NOR		1	1111		0			0		0	1	0			0		0
XOR		1	1110		0			0		0	1	0			0		0
MULT		1	"--00"			0						0			0		0
MFHI					1		1			0	1	0			0		0
MFLO					1		0			0	1	0			0		0
MTHI						1	1					0			0		0
MTLO						1	0					0			0		0
SLL		1	0	0	0		0	0		0	1	0			0		0
SRL		1	10	0	0			0		0	1	0			0		0
SRA		1	11	0	0			0		0	1	0			0		0
SLLV		1	0	1	0		0	0		0	1	0			0		0
SRLV		1	10	1	0			0		0	1	0			0		0
SRAV		1	11	1	0			0		0	1	0			0		0
LUI	-	0	0	0	0		1	0		0	0	0			0		0
SLTI	1	0	110		0			0		0	0	0			0		0
SLTIU	1	0	111		0			0		0	0	0			0		0
SLT		1	110		0			0		0	1	0			0		0
SLT		1	111		0			0		0	1	0			0		0

U																	
BEQ	1	1	1010									01	0		0		0
BNE	1	1	1010									01	1		0		0
BLEZ	1	1	1010									10	0		0		0
BGTZ	1	1	1010									10	1		0		0
BLTZ	1	1	1010									11	0	1	0		0
BGEZ	1	1	1010									11	1	1	0		0
JR															1	0	0
JALR										1	1				1	0	0
J															1	1	0
JAL										1	1				1	1	0
TEST																	1

Block διάγραμμα



```
library ieee;
use ieee.std_logic_1164.all;

-- Most outputs are registered to minimize critical path and clock period.
-- Those that cannot be registered are the ones needed in the ID stage from
-- the register file or the sign extension units (RorI, RTZero, SorZ).

entity control_comb is
    generic(mult_pipe    : boolean := true);
    port(   clk          : in  std_logic;
            rst          : in  std_logic;
            OPCODE       : in  std_logic_vector(5 downto 0);
            FUNCT        : in  std_logic_vector(5 downto 0);
            RT           : in  std_logic_vector(4 downto 0);
            SorZ         : out std_logic;
            BorI         : out std_logic;
            ALUop        : out std_logic_vector(3 downto 0);
            sv           : out std_logic;
            MF           : out std_logic;
            MT           : out std_logic;
            HIorLO       : out std_logic;
            DMorALU      : out std_logic;
            DMWT         : out std_logic_vector(2 downto 0);
            Link         : out std_logic;
            RorI         : out std_logic;
            BranchType   : out std_logic_vector(1 downto 0);
            NEorEQ       : out std_logic;
            RTZero       : out std_logic;
            Jump         : out std_logic;
            JumpPSD      : out std_logic;
            TestMult     : out std_logic);
end control_comb;

architecture Behavioral of control_comb is

    -- OPCODE definition as constants
    constant RTYPE   : std_logic_vector(5 downto 0) := "000000"; -- 0x00
    constant BLTZ    : std_logic_vector(5 downto 0) := "000001"; -- 0x01
    --constant BGEZ    : std_logic_vector(5 downto 0) := "000001"; -- 0x01
    constant J       : std_logic_vector(5 downto 0) := "000010"; -- 0x02
    constant JAL     : std_logic_vector(5 downto 0) := "000011"; -- 0x03
    constant BEQ     : std_logic_vector(5 downto 0) := "000100"; -- 0x04
    constant BNE     : std_logic_vector(5 downto 0) := "000101"; -- 0x05
    constant BLEZ    : std_logic_vector(5 downto 0) := "000110"; -- 0x06
    constant BGTZ    : std_logic_vector(5 downto 0) := "000111"; -- 0x07
    constant ADDI    : std_logic_vector(5 downto 0) := "001000"; -- 0x08
    constant ADDIU   : std_logic_vector(5 downto 0) := "001001"; -- 0x09
    constant SLTI    : std_logic_vector(5 downto 0) := "001010"; -- 0x0A
    constant SLTIU   : std_logic_vector(5 downto 0) := "001011"; -- 0x0B
    constant ANDI    : std_logic_vector(5 downto 0) := "001100"; -- 0x0C
    constant ORI     : std_logic_vector(5 downto 0) := "001101"; -- 0x0D
    constant XORI    : std_logic_vector(5 downto 0) := "001110"; -- 0x0E
    constant LUI     : std_logic_vector(5 downto 0) := "001111"; -- 0x0F
    constant LB      : std_logic_vector(5 downto 0) := "100000"; -- 0x20
    constant LH      : std_logic_vector(5 downto 0) := "100001"; -- 0x21
    constant LW      : std_logic_vector(5 downto 0) := "100011"; -- 0x23
    constant LBU     : std_logic_vector(5 downto 0) := "100100"; -- 0x24
    constant LHU     : std_logic_vector(5 downto 0) := "100101"; -- 0x25
    constant SB      : std_logic_vector(5 downto 0) := "101000"; -- 0x28
```

```
    constant SH     : std_logic_vector(5 downto 0) := "101001"; -- 0x29
    constant SW     : std_logic_vector(5 downto 0) := "101011"; -- 0x2B
    constant TEST   : std_logic_vector(5 downto 0) := "110000"; -- 0x30

    -- FUNCT definition as constants
    constant SLLR   : std_logic_vector(5 downto 0) := "000000"; -- 0x00
    constant SRLR   : std_logic_vector(5 downto 0) := "000010"; -- 0x02
    constant SRAR   : std_logic_vector(5 downto 0) := "000011"; -- 0x03
    constant SLLVR  : std_logic_vector(5 downto 0) := "000100"; -- 0x04
    constant SRLVR  : std_logic_vector(5 downto 0) := "000110"; -- 0x06
    constant SRAVR  : std_logic_vector(5 downto 0) := "000111"; -- 0x07
    constant JR     : std_logic_vector(5 downto 0) := "001000"; -- 0x08
    constant JALR   : std_logic_vector(5 downto 0) := "001001"; -- 0x09
    constant MFHI   : std_logic_vector(5 downto 0) := "010000"; -- 0x10
    constant MTHI   : std_logic_vector(5 downto 0) := "010001"; -- 0x11
    constant MFLO   : std_logic_vector(5 downto 0) := "010010"; -- 0x12
    constant MTLO   : std_logic_vector(5 downto 0) := "010011"; -- 0x13
    constant MULTR  : std_logic_vector(5 downto 0) := "011000"; -- 0x18
    constant ADDR   : std_logic_vector(5 downto 0) := "100000"; -- 0x20
    constant ADDRU  : std_logic_vector(5 downto 0) := "100001"; -- 0x21
    constant SUBR   : std_logic_vector(5 downto 0) := "100010"; -- 0x22
    constant SUBRU  : std_logic_vector(5 downto 0) := "100011"; -- 0x23
    constant ANDR   : std_logic_vector(5 downto 0) := "100100"; -- 0x24
    constant ORR    : std_logic_vector(5 downto 0) := "100101"; -- 0x25
    constant XORR   : std_logic_vector(5 downto 0) := "100110"; -- 0x26
    constant NORR   : std_logic_vector(5 downto 0) := "100111"; -- 0x27
    constant SLTR   : std_logic_vector(5 downto 0) := "101010"; -- 0x2A
    constant SLTRU  : std_logic_vector(5 downto 0) := "101011"; -- 0x2B

begin

    SorZ    <= '0' when OPCODE = ANDI or OPCODE = ORI or OPCODE = XORI else '1';
    RorI    <= '1' when OPCODE = "000000" else '0';
    RTZero  <= '1' when OPCODE = "000001" else '-';

    cntr_comb: process (clk, rst, OPCODE, FUNCT, RT)
    begin

        if(rst = '1') then

            -- OUTPUT initialization

            -- SorZ         <= '0';
            BorI        <= '0';
            ALUop       <= "0000";
            sv          <= '0';
            MF          <= '0';
            MT          <= '0';
            HIorLO      <= '0';
            DMorALU     <= '0';
            DMWT        <= "000";
            Link        <= '0';
            -- RorI         <= '0';
            BranchType  <=  "00";
            NEorEQ      <= '0';
            -- RTZero       <= '0';
            Jump        <= '0';
            JumpPSD     <= '0';
            TestMult    <= '0';

        elsif(clk'event and clk = '1') then

            case OPCODE is
```

```
when TEST =>
    -- SorZ         <= '-';
    BorI         <= '-';
    ALUop        <= "----";
    sv           <= '-';
    MF           <= '-';
    MT           <= '-';
    HIorLO       <= '-';
    DMorALU      <= '-';
    DMWT         <= "---";
    Link         <= '-';
    -- RorI         <= '-';
    BranchType   <=  "--";
    NEorEQ       <= '-';
    -- RTZero       <= '-';
    Jump         <= '-';
    JumpPSD      <= '-';
    TestMult     <= '1';

when LW =>
    -- SorZ         <= '1';
    BorI         <= '0';
    ALUop        <= "1001";
    sv           <= '-';
    MF           <= '0';
    MT           <= '-';
    HIorLO       <= '-';
    DMorALU      <= '1';
    DMWT         <= "100";
    Link         <= '0';
    -- RorI         <= '0';
    BranchType   <= "00";
    NEorEQ       <= '-';
    -- RTZero       <= '-';
    Jump         <= '0';
    JumpPSD      <= '-';
    TestMult     <= '0';

when LH =>
    -- SorZ         <= '1';
    BorI         <= '0';
    ALUop        <= "1001";
    sv           <= '-';
    MF           <= '0';
    MT           <= '-';
    HIorLO       <= '-';
    DMorALU      <= '1';
    DMWT         <= "011";
    Link         <= '0';
    -- RorI         <= '0';
    BranchType   <= "00";
    NEorEQ       <= '-';
    -- RTZero       <= '-';
    Jump         <= '0';
    JumpPSD      <= '-';
    TestMult     <= '0';

when LHU =>
    -- SorZ         <= '1';
    BorI         <= '0';
    ALUop        <= "1001";
    sv           <= '-';
```

```
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '1';
            DMWT        <= "010";
            Link        <= '0';
            -- RorI        <= '0';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when LB =>
            -- SorZ        <= '1';
            BorI        <= '0';
            ALUop       <= "1001";
            sv          <= '-';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '1';
            DMWT        <= "001";
            Link        <= '0';
            -- RorI        <= '0';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when LBU =>
            -- SorZ        <= '1';
            BorI        <= '0';
            ALUop       <= "1001";
            sv          <= '-';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '1';
            DMWT        <= "000";
            Link        <= '0';
            -- RorI        <= '0';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when SW =>
            -- SorZ        <= '1';
            BorI        <= '0';
            ALUop       <= "1001";
            sv          <= '-';
            MF          <= '-';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '-';
            DMWT        <= "100";
            Link        <= '-';
```

```
            -- RorI        <= '-';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when SH =>
            -- SorZ        <= '1';
            BorI        <= '0';
            ALUop       <= "1001";
            sv          <= '-';
            MF          <= '-';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '-';
            DMWT        <= "011";
            Link        <= '-';
            -- RorI        <= '-';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when SB =>
            -- SorZ        <= '1';
            BorI        <= '0';
            ALUop       <= "1001";
            sv          <= '-';
            MF          <= '-';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '-';
            DMWT        <= "001";
            Link        <= '-';
            -- RorI        <= '-';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when BLTZ =>
        -- when BGEZ =>
            -- SorZ        <= '1';
            BorI        <= '1';
            ALUop       <= "1010";
            sv          <= '-';
            MF          <= '-';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '-';
            DMWT        <= "---";
            Link        <= '-';
            -- RorI        <= '-';
            BranchType  <= "11";
            NEorEQ      <= RT(0);
            -- RTZero      <= '1';
            Jump        <= '0';
```

```
            JumpPSD     <= '-';
            TestMult    <= '0';

        when BLEZ =>
            -- SorZ        <= '1';
            BorI        <= '1';
            ALUop       <= "1010";
            sv          <= '-';
            MF          <= '-';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '-';
            DMWT        <= "---";
            Link        <= '-';
            -- RorI        <= '-';
            BranchType  <= "10";
            NEorEQ      <= '0';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when BGTZ =>
            -- SorZ        <= '1';
            BorI        <= '1';
            ALUop       <= "1010";
            sv          <= '-';
            MF          <= '-';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '-';
            DMWT        <= "---";
            Link        <= '-';
            -- RorI        <= '-';
            BranchType  <= "10";
            NEorEQ      <= '1';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when BEQ =>
            -- SorZ        <= '1';
            BorI        <= '1';
            ALUop       <= "1010";
            sv          <= '-';
            MF          <= '-';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '-';
            DMWT        <= "---";
            Link        <= '-';
            -- RorI        <= '-';
            BranchType  <= "01";
            NEorEQ      <= '0';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when BNE =>
            -- SorZ        <= '1';
            BorI        <= '1';
```

```
            ALUop       <= "1010";
            sv          <= '-';
            MF          <= '-';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '-';
            DMWT        <= "---";
            Link        <= '-';
            -- RorI        <= '-';
            BranchType  <= "01";
            NEorEQ      <= '1';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when ADDI =>
            -- SorZ        <= '1';
            BorI        <= '0';
            ALUop       <= "1000";
            sv          <= '-';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '0';
            DMWT        <= "---";
            Link        <= '0';
            -- RorI        <= '0';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when ADDIU =>
            -- SorZ        <= '1';
            BorI        <= '0';
            ALUop       <= "1001";
            sv          <= '-';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '0';
            DMWT        <= "---";
            Link        <= '0';
            -- RorI        <= '0';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when SLTI =>
            -- SorZ        <= '1';
            BorI        <= '0';
            ALUop       <= "0110";
            sv          <= '-';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '0';
```

```
            DMWT        <= "---";
            Link        <= '0';
            -- RorI        <= '0';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when SLTIU =>
            -- SorZ        <= '1';
            BorI        <= '0';
            ALUop       <= "0111";
            sv          <= '-';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '0';
            DMWT        <= "---";
            Link        <= '0';
            -- RorI        <= '0';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when ANDI =>
            -- SorZ        <= '0';
            BorI        <= '0';
            ALUop       <= "1100";
            sv          <= '-';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '0';
            DMWT        <= "---";
            Link        <= '0';
            -- RorI        <= '0';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when ORI =>
            -- SorZ        <= '0';
            BorI        <= '0';
            ALUop       <= "1101";
            sv          <= '-';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '0';
            DMWT        <= "---";
            Link        <= '0';
            -- RorI        <= '0';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
```

```
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when XORI =>
            -- SorZ        <= '0';
            BorI        <= '0';
            ALUop       <= "1110";
            sv          <= '-';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '0';
            DMWT        <= "---";
            Link        <= '0';
            -- RorI        <= '0';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when LUI =>
            -- SorZ        <= '-';
            BorI        <= '0';
            ALUop       <= "0000";
            sv          <= '0';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '1';
            DMorALU     <= '0';
            DMWT        <= "---";
            Link        <= '0';
            -- RorI        <= '0';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when J =>
            -- SorZ        <= '-';
            BorI        <= '-';
            ALUop       <= "----";
            sv          <= '-';
            MF          <= '-';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '-';
            DMWT        <= "---";
            Link        <= '-';
            -- RorI        <= '-';
            BranchType  <= "--";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '1';
            JumpPSD     <= '1';
            TestMult    <= '0';

        when JAL =>
            -- SorZ        <= '-';
```

```
            BorI        <= '-';
            ALUop       <= "----";
            sv          <= '-';
            MF          <= '-';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '-';
            DMWT        <= "---";
            Link        <= '1';
            -- RorI        <= '1';
            BranchType  <= "--";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '1';
            JumpPSD     <= '1';
            TestMult    <= '0';

        when RTYPE =>

            case FUNCT is

                when ADDR =>
                    -- SorZ        <= '-';
                    BorI        <= '1';
                    ALUop       <= "1000";
                    sv          <= '-';
                    MF          <= '0';
                    MT          <= '-';
                    HIorLO      <= '-';
                    DMorALU     <= '0';
                    DMWT        <= "---";
                    Link        <= '0';
                    -- RorI        <= '1';
                    BranchType  <= "00";
                    NEorEQ      <= '-';
                    -- RTZero      <= '-';
                    Jump        <= '0';
                    JumpPSD     <= '-';
                    TestMult    <= '0';

                when ADDRU =>
                    -- SorZ        <= '-';
                    BorI        <= '1';
                    ALUop       <= "1001";
                    sv          <= '-';
                    MF          <= '0';
                    MT          <= '-';
                    HIorLO      <= '-';
                    DMorALU     <= '0';
                    DMWT        <= "---";
                    Link        <= '0';
                    -- RorI        <= '1';
                    BranchType  <= "00";
                    NEorEQ      <= '-';
                    -- RTZero      <= '-';
                    Jump        <= '0';
                    JumpPSD     <= '-';
                    TestMult    <= '0';

                when SUBR =>
                    -- SorZ        <= '-';
                    BorI        <= '1';
                    ALUop       <= "1010";
```

```
            sv          <= '-';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '0';
            DMWT        <= "---";
            Link        <= '0';
            -- RorI         <= '1';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero       <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when SUBRU =>
            -- SorZ         <= '-';
            BorI        <= '1';
            ALUop       <= "1011";
            sv          <= '-';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '0';
            DMWT        <= "---";
            Link        <= '0';
            -- RorI         <= '1';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero       <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when MULTR =>
            -- SorZ         <= '-';
            BorI        <= '1';
            ALUop       <= "--00";
            sv          <= '-';
            MF          <= '-';
            MT          <= '0';
            HIorLO      <= '-';
            DMorALU     <= '-';
            DMWT        <= "---";
            Link        <= '-';
            -- RorI         <= '-';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero       <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when MFHI =>
            -- SorZ         <= '-';
            BorI        <= '-';
            ALUop       <= "----";
            sv          <= '-';
            MF          <= '1';
            MT          <= '-';
            HIorLO      <= '1';
            DMorALU     <= '-';
            DMWT        <= "---";
```

```
            Link        <= '0';
            -- RorI        <= '1';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when MTHI =>
            -- SorZ        <= '-';
            BorI        <= '-';
            ALUop       <= "----";
            sv          <= '-';
            MF          <= '-';
            MT          <= '1';
            HIorLO      <= '1';
            DMorALU     <= '-';
            DMWT        <= "---";
            Link        <= '-';
            -- RorI        <= '-';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when MFLO =>
            -- SorZ        <= '-';
            BorI        <= '-';
            ALUop       <= "----";
            sv          <= '-';
            MF          <= '1';
            MT          <= '-';
            HIorLO      <= '0';
            DMorALU     <= '-';
            DMWT        <= "---";
            Link        <= '0';
            -- RorI        <= '1';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when MTLO =>
            -- SorZ        <= '-';
            BorI        <= '-';
            ALUop       <= "----";
            sv          <= '-';
            MF          <= '-';
            MT          <= '1';
            HIorLO      <= '0';
            DMorALU     <= '-';
            DMWT        <= "---";
            Link        <= '-';
            -- RorI        <= '-';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
```

```
            JumpPSD     <= '-';
            TestMult    <= '0';

        when ANDR =>
            -- SorZ        <= '-';
            BorI        <= '1';
            ALUop       <= "1100";
            sv          <= '-';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '0';
            DMWT        <= "---";
            Link        <= '0';
            -- RorI        <= '1';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when ORR =>
            -- SorZ        <= '-';
            BorI        <= '1';
            ALUop       <= "1101";
            sv          <= '-';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '0';
            DMWT        <= "---";
            Link        <= '0';
            -- RorI        <= '1';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when XORR =>
            -- SorZ        <= '-';
            BorI        <= '1';
            ALUop       <= "1110";
            sv          <= '-';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '0';
            DMWT        <= "---";
            Link        <= '0';
            -- RorI        <= '1';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when NORR =>
            -- SorZ        <= '-';
            BorI        <= '1';
```

```
            ALUop       <= "1111";
            sv          <= '-';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '0';
            DMWT        <= "---";
            Link        <= '0';
            -- RorI        <= '1';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when SLTR =>
            -- SorZ        <= '-';
            BorI        <= '1';
            ALUop       <= "0110";
            sv          <= '-';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '0';
            DMWT        <= "---";
            Link        <= '0';
            -- RorI        <= '1';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when SLTRU =>
            -- SorZ        <= '-';
            BorI        <= '1';
            ALUop       <= "0111";
            sv          <= '-';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '0';
            DMWT        <= "---";
            Link        <= '0';
            -- RorI        <= '1';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when SLLR =>
            -- SorZ        <= '-';
            BorI        <= '1';
            ALUop       <= "0000";
            sv          <= '0';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '0';
            DMorALU     <= '0';
```

```
            DMWT        <= "---";
            Link        <= '0';
            -- RorI        <= '1';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when SRLR =>
            -- SorZ        <= '-';
            BorI        <= '1';
            ALUop       <= "0010";
            sv          <= '0';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '0';
            DMWT        <= "---";
            Link        <= '0';
            -- RorI        <= '1';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when SRAR =>
            -- SorZ        <= '-';
            BorI        <= '1';
            ALUop       <= "0011";
            sv          <= '0';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '0';
            DMWT        <= "---";
            Link        <= '0';
            -- RorI        <= '1';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when SLLVR =>
            -- SorZ        <= '-';
            BorI        <= '1';
            ALUop       <= "0000";
            sv          <= '1';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '0';
            DMorALU     <= '0';
            DMWT        <= "---";
            Link        <= '0';
            -- RorI        <= '1';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
```

```
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when SRLVR =>
            -- SorZ        <= '-';
            BorI        <= '1';
            ALUop       <= "0010";
            sv          <= '1';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '0';
            DMWT        <= "---";
            Link        <= '0';
            -- RorI        <= '1';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when SRAVR =>
            -- SorZ        <= '-';
            BorI        <= '1';
            ALUop       <= "0011";
            sv          <= '1';
            MF          <= '0';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '0';
            DMWT        <= "---";
            Link        <= '0';
            -- RorI        <= '1';
            BranchType  <= "00";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '0';
            JumpPSD     <= '-';
            TestMult    <= '0';

        when JR =>
            -- SorZ        <= '-';
            BorI        <= '-';
            ALUop       <= "----";
            sv          <= '-';
            MF          <= '-';
            MT          <= '-';
            HIorLO      <= '-';
            DMorALU     <= '-';
            DMWT        <= "---";
            Link        <= '-';
            -- RorI        <= '-';
            BranchType  <= "--";
            NEorEQ      <= '-';
            -- RTZero      <= '-';
            Jump        <= '1';
            JumpPSD     <= '0';
            TestMult    <= '0';

        when JALR =>
            -- SorZ        <= '-';
```

```
                        BorI        <= '-';
                        ALUop       <= "----";
                        sv          <= '-';
                        MF          <= '-';
                        MT          <= '-';
                        HIorLO      <= '-';
                        DMorALU     <= '-';
                        DMWT        <= "---";
                        Link        <= '1';
                        -- RorI       <= '1';
                        BranchType  <= "--";
                        NEorEQ      <= '-';
                        -- RTZero     <= '-';
                        Jump        <= '1';
                        JumpPSD     <= '0';
                        TestMult    <= '0';

                    when others =>
                        -- SorZ       <= '-';
                        BorI        <= '-';
                        ALUop       <= "----";
                        sv          <= '-';
                        MF          <= '-';
                        MT          <= '-';
                        HIorLO      <= '-';
                        DMorALU     <= '-';
                        DMWT        <= "---";
                        Link        <= '-';
                        -- RorI       <= '-';
                        BranchType  <= "--";
                        NEorEQ      <= '-';
                        -- RTZero     <= '-';
                        Jump        <= '-';
                        JumpPSD     <= '-';
                        TestMult    <= '-';

                end case;

            when others =>
                -- SorZ       <= '-';
                BorI        <= '-';
                ALUop       <= "----";
                sv          <= '-';
                MF          <= '-';
                MT          <= '-';
                HIorLO      <= '-';
                DMorALU     <= '-';
                DMWT        <= "---";
                Link        <= '-';
                -- RorI       <= '-';
                BranchType  <= "--";
                NEorEQ      <= '-';
                -- RTZero     <= '-';
                Jump        <= '-';
                JumpPSD     <= '-';
                TestMult    <= '-';

        end case;
    end if;
end process;

end Behavioral;
```

Σύγχρονη ακολουθιακή μονάδα

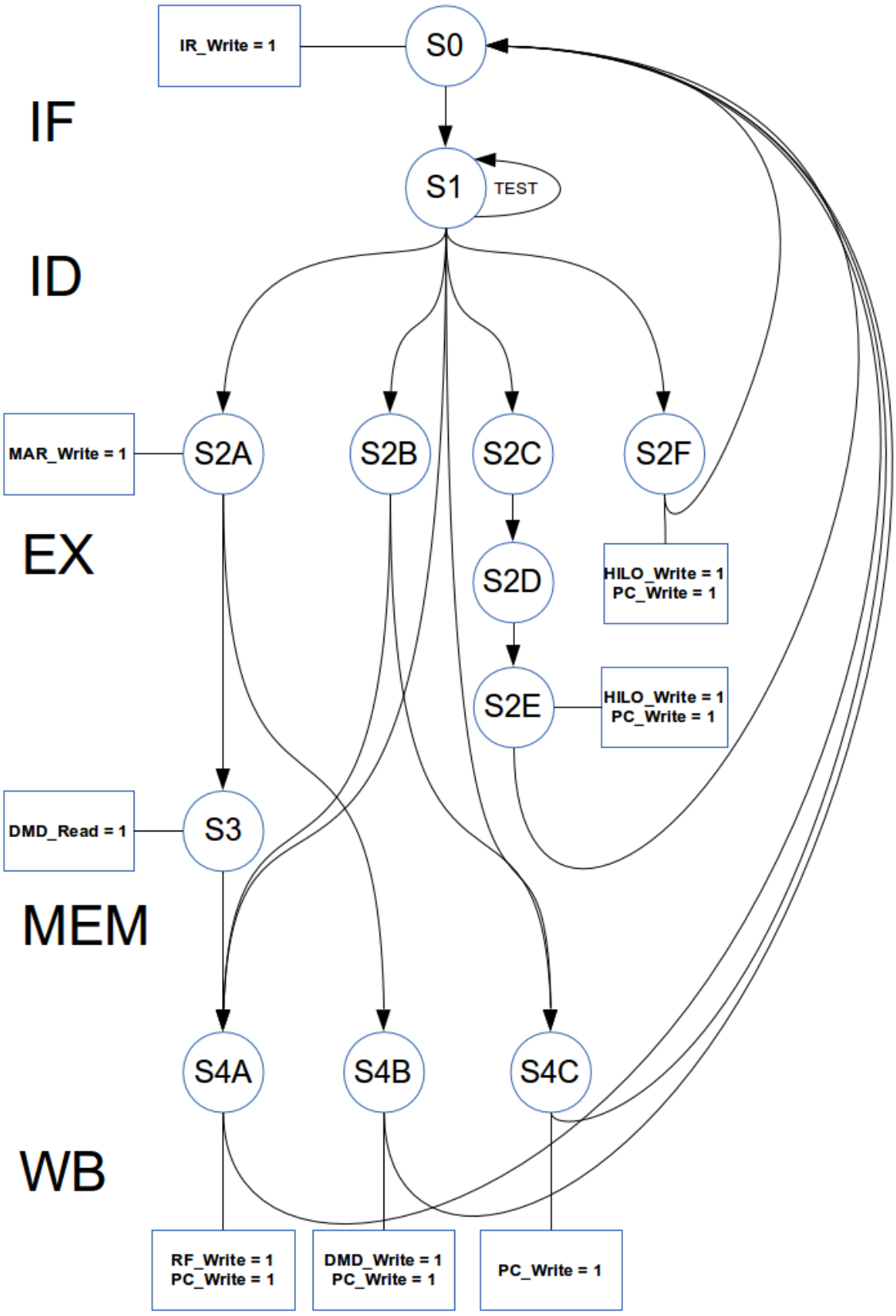
Γενική περιγραφή

Η μονάδα αυτή είναι καθαρά συνδυαστική και περιλαμβάνει μια μηχανή πεπερασμένων καταστάσεων (FSM) τύπου Moore υλοποιημένη με 3 process. Στην πραγματικότητα έχουμε αφαιρέσει το τρίτο process για τον καθορισμό των εξόδων ανάλογα την κατάσταση και το έχουμε αντικαταστήσει με συνδυαστική λογική χωρίς process. Όλα αυτά είναι σύμφωνα με τα σωστά πρότυπα συγγραφής VHDL και συνιστούνται κατά τη συγγραφή μιας FSM τύπου Moore (ή και Mealy).

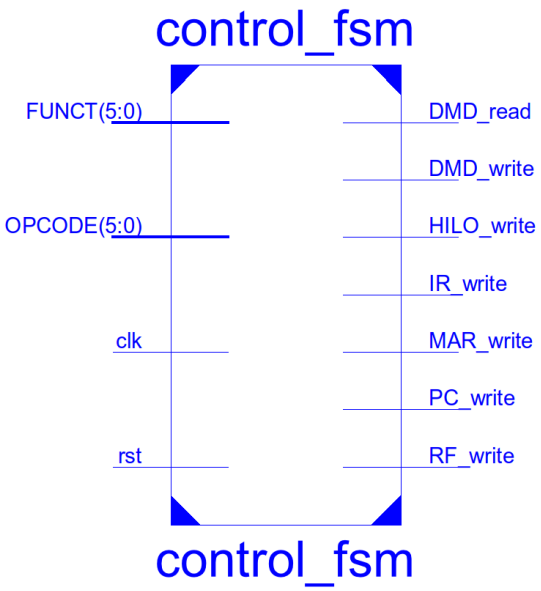
Το διάγραμμα καταστάσεων της FSM φαίνεται παρακάτω και δείχνει κατανοητά όλα τα στάδια τα οποία πρέπει να περάσει κάθε εντολή κατά τη διάρκεια της εκτέλεσης της. Τα πρώτα δύο στάδια (IF, ID) είναι κοινά για όλες τις εντολές οπότε και πρατηρούμε οτι δεν υπάρχει κάποια διακλάδωση σε αυτά.

Τέλος πρέπει να αναφέρουμε οτι μόνο για την εντολή MULT το πλήθος των καταστάσεων (άρα και των κύκλων ρολογιού) είναι μεταβλητό και εξαρτάται από την generic παράμετρο mult_pipe η οποία ορίζει αν ο πολλαπλασιαστής μας είναι υλοποιημένος με διοχέτευση όπως περιγράφτηκε στην αντίστοιχη ενότητα.

Διάγραμμα καταστάσεων



Block διάγραμμα



VHDL κώδικας

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity control_fsm is
    generic(mult_pipe    : boolean := true);
    port(    clk         : in  std_logic;
             rst         : in  std_logic;
             OPCODE      : in  std_logic_vector(5 downto 0);
             FUNCT       : in  std_logic_vector(5 downto 0);
             PC_write    : out std_logic;
             IR_write    : out std_logic;
             MAR_write   : out std_logic;
             DMD_read    : out std_logic;
             DMD_write   : out std_logic;
             RF_write    : out std_logic;
             HILO_write    : out std_logic);
end control_fsm;

architecture Behavioral of control_fsm is

    -- state definition
    type control_states is (S0, S1, S2A, S2B, S2C, S2D, S2E, S2F, S3, S4A, S4B,
S4C);
    signal current_state, next_state : control_states;

    -- OPCODE definition as constants
    constant RTYPE  : std_logic_vector(5 downto 0) := "000000"; -- 0x00
    constant BLTZ   : std_logic_vector(5 downto 0) := "000001"; -- 0x01
    --constant BGEZ   : std_logic_vector(5 downto 0) := "000001"; -- 0x01   --
Don't need it because it has the same opcode
    constant J      : std_logic_vector(5 downto 0) := "000010"; -- 0x02
```

```
    constant JAL    : std_logic_vector(5 downto 0) := "000011"; -- 0x03
    constant BEQ    : std_logic_vector(5 downto 0) := "000100"; -- 0x04
    constant BNE    : std_logic_vector(5 downto 0) := "000101"; -- 0x05
    constant BLEZ   : std_logic_vector(5 downto 0) := "000110"; -- 0x06
    constant BGTZ   : std_logic_vector(5 downto 0) := "000111"; -- 0x07
    constant ADDI   : std_logic_vector(5 downto 0) := "001000"; -- 0x08
    constant ADDIU  : std_logic_vector(5 downto 0) := "001001"; -- 0x09
    constant SLTI   : std_logic_vector(5 downto 0) := "001010"; -- 0x0A
    constant SLTIU  : std_logic_vector(5 downto 0) := "001011"; -- 0x0B
    constant ANDI   : std_logic_vector(5 downto 0) := "001100"; -- 0x0C
    constant ORI    : std_logic_vector(5 downto 0) := "001101"; -- 0x0D
    constant XORI   : std_logic_vector(5 downto 0) := "001110"; -- 0x0E
    constant LUI    : std_logic_vector(5 downto 0) := "001111"; -- 0x0F
    constant LB     : std_logic_vector(5 downto 0) := "100000"; -- 0x20
    constant LH     : std_logic_vector(5 downto 0) := "100001"; -- 0x21
    constant LW     : std_logic_vector(5 downto 0) := "100011"; -- 0x23
    constant LBU    : std_logic_vector(5 downto 0) := "100100"; -- 0x24
    constant LHU    : std_logic_vector(5 downto 0) := "100101"; -- 0x25
    constant SB     : std_logic_vector(5 downto 0) := "101000"; -- 0x28
    constant SH     : std_logic_vector(5 downto 0) := "101001"; -- 0x29
    constant SW     : std_logic_vector(5 downto 0) := "101011"; -- 0x2B
    constant TEST   : std_logic_vector(5 downto 0) := "110000"; -- 0x30

    -- FUNCT definition as constants
    constant SLLR   : std_logic_vector(5 downto 0) := "000000"; -- 0x00
    constant SRLR   : std_logic_vector(5 downto 0) := "000010"; -- 0x02
    constant SRAR   : std_logic_vector(5 downto 0) := "000011"; -- 0x03
    constant SLLVR  : std_logic_vector(5 downto 0) := "000100"; -- 0x04
    constant SRLVR  : std_logic_vector(5 downto 0) := "000110"; -- 0x06
    constant SRAVR  : std_logic_vector(5 downto 0) := "000111"; -- 0x07
    constant JR     : std_logic_vector(5 downto 0) := "001000"; -- 0x08
    constant JALR   : std_logic_vector(5 downto 0) := "001001"; -- 0x09
    constant MFHI   : std_logic_vector(5 downto 0) := "010000"; -- 0x10
    constant MTHI   : std_logic_vector(5 downto 0) := "010001"; -- 0x11
    constant MFLO   : std_logic_vector(5 downto 0) := "010010"; -- 0x12
    constant MTLO   : std_logic_vector(5 downto 0) := "010011"; -- 0x13
    constant MULTR  : std_logic_vector(5 downto 0) := "011000"; -- 0x18
    constant ADDR   : std_logic_vector(5 downto 0) := "100000"; -- 0x20
    constant ADDRU  : std_logic_vector(5 downto 0) := "100001"; -- 0x21
    constant SUBR   : std_logic_vector(5 downto 0) := "100010"; -- 0x22
    constant SUBRU  : std_logic_vector(5 downto 0) := "100011"; -- 0x23
    constant ANDR   : std_logic_vector(5 downto 0) := "100100"; -- 0x24
    constant ORR    : std_logic_vector(5 downto 0) := "100101"; -- 0x25
    constant XORR   : std_logic_vector(5 downto 0) := "100110"; -- 0x26
    constant NORR   : std_logic_vector(5 downto 0) := "100111"; -- 0x27
    constant SLTR   : std_logic_vector(5 downto 0) := "101010"; -- 0x2A
    constant SLTRU  : std_logic_vector(5 downto 0) := "101011"; -- 0x2B

    signal mult_counter : std_logic_vector(1 downto 0);
    signal mult_cycles  : std_logic_vector(1 downto 0);

begin

    -- Multiplier cycles

    pipelined: if (mult_pipe = true) generate

        -- Pipelined multiplier (4 clock cycles latency)

        mult_cycles <= "11";

    end generate;
```

```
    normal: if(mult_pipe = false) generate

         -- Normal Multiplier (1 clock cycle latency)

         mult_cycles <= "00";

    end generate;

    -- common synchronous process for all FSMs
    SYNCHR: process (clk, rst)
    begin

        if(rst = '1') then
            mult_counter    <= (others => '0');

            current_state   <= S0; -- initial state

        elsif(clk'event and clk = '1') then

            case current_state is

                when S0 =>      mult_counter <= (others => '0');

                when S2D =>     if(mult_counter = mult_cycles) then
                                    mult_counter <= (others => '0');
                                else
                                    mult_counter <=
std_logic_vector(unsigned(mult_counter) + 1);
                                end if;

                when others =>  null;

                end case;

            current_state <= next_state;
        end if;

    end process;

    -- asynchronous process to create output logic and next state logic
    ASYNCHR: process (current_state, OPCODE, FUNCT, mult_counter)
    begin

        -- Next state is by default the current_state
        next_state <= current_state;

        case current_state is

            when S0 =>      -- IF

                next_state <= S1;

            when S1 =>      -- ID

                case OPCODE is

                    when BLTZ   =>  next_state  <= S2B;
                    -- when BGEZ   =>  next_state  <= S2B;
                    when J      =>  next_state  <= S4C;
                    when JAL    =>  next_state  <= S4A;
                    when BEQ    =>  next_state  <= S2B;
                    when BNE    =>  next_state  <= S2B;
                    when BLEZ   =>  next_state  <= S2B;
```

```
                when BGTZ   =>  next_state  <= S2B;
                when ADDI   =>  next_state  <= S2B;
                when ADDIU  =>  next_state  <= S2B;
                when SLTI   =>  next_state  <= S2B;
                when SLTIU  =>  next_state  <= S2B;
                when ANDI   =>  next_state  <= S2B;
                when ORI    =>  next_state  <= S2B;
                when XORI   =>  next_state  <= S2B;
                when LUI    =>  next_state  <= S2B;
                when LW     =>  next_state  <= S2A;
                when LH     =>  next_state  <= S2A;
                when LHU    =>  next_state  <= S2A;
                when LB     =>  next_state  <= S2A;
                when LBU    =>  next_state  <= S2A;
                when SW     =>  next_state  <= S2A;
                when SH     =>  next_state  <= S2A;
                when SB     =>  next_state  <= S2A;
                when TEST   =>  next_state  <= S1;
                when RTYPE =>

                    case FUNCT is

                        when SLLR   =>  next_state  <= S2B;
                        when SRLR   =>  next_state  <= S2B;
                        when SRAR   =>  next_state  <= S2B;
                        when SLLVR  =>  next_state  <= S2B;
                        when SRLVR  =>  next_state  <= S2B;
                        when SRAVR  =>  next_state  <= S2B;
                        when JR     =>  next_state  <= S4C;
                        when JALR   =>  next_state  <= S4A;
                        when MFHI   =>  next_state  <= S4A;
                        when MFLO   =>  next_state  <= S4A;
                        when MTHI   =>  next_state  <= S2F;
                        when MTLO   =>  next_state  <= S2F;
                        when MULTR  =>  next_state  <= S2C;
                        when ADDR   =>  next_state  <= S2B;
                        when ADDRU  =>  next_state  <= S2B;
                        when SUBR   =>  next_state  <= S2B;
                        when SUBRU  =>  next_state  <= S2B;
                        when ANDR   =>  next_state  <= S2B;
                        when ORR    =>  next_state  <= S2B;
                        when XORR   =>  next_state  <= S2B;
                        when NORR   =>  next_state  <= S2B;
                        when SLTR   =>  next_state  <= S2B;
                        when SLTRU  =>  next_state  <= S2B;
                        when others =>  next_state  <= S0;

                    end case;

                when others =>  next_state  <= S0;

            end case;

        when S2A =>       -- EX (LW, LH, LHU, LB, LBU & SW, SH, SB)

            case OPCODE is

                when LW  =>     next_state  <= S3;
                when LH  =>     next_state  <= S3;
                when LHU =>     next_state  <= S3;
                when LB  =>     next_state  <= S3;
                when LBU =>     next_state  <= S3;
                when SW  =>     next_state  <= S4B;
```

```
                    when SH  =>     next_state  <= S4B;
                    when SB  =>     next_state  <= S4B;
                    when others =>  next_state  <= S0;

                end case;

            when S2B =>      -- EX (Normal)

                case OPCODE is

                    when BLTZ   =>  next_state  <= S4C;
    --              when BGEZ   =>  next_state  <= S4C;
                    when BEQ    =>  next_state  <= S4C;
                    when BNE    =>  next_state  <= S4C;
                    when BLEZ   =>  next_state  <= S4C;
                    when BGTZ   =>  next_state  <= S4C;
                    when ADDI   =>  next_state  <= S4A;
                    when ADDIU  =>  next_state  <= S4A;
                    when SLTI   =>  next_state  <= S4A;
                    when SLTIU  =>  next_state  <= S4A;
                    when ANDI   =>  next_state  <= S4A;
                    when ORI    =>  next_state  <= S4A;
                    when XORI   =>  next_state  <= S4A;
                    when LUI    =>  next_state  <= S4A;
                    when RTYPE  =>  next_state  <= S4A;
                    when others =>  next_state  <= S0;

                end case;

            when S2C =>     next_state <= S2D;

            when S2D =>    if(mult_counter = mult_cycles) then      -- EX (MULT)
                               next_state <= S2E;                    -- 4 cycles
here for pipelined multiplier
                           else                                      -- 1 cycle
here for normal multiplier
                               next_state  <= S2D;
                           end if;

            when S2E =>     next_state  <= S0;  -- EX & WB (MULT)

            when S2F =>     next_state  <= S0;  -- EX & WB (MTHI, MTLO)

            when S3 =>      next_state  <= S4A; -- MEM (LW, LH, LHU, LB, LBU)

            when S4A =>     next_state  <= S0;  -- WB (Normal)
            when S4B =>     next_state  <= S0;  -- WB (SW, SH, SB)
            when S4C =>     next_state  <= S0;  -- WB (Jumps without link and
Branches)

            -- Other case not needed because we have a path for all the states
            -- when others =>  next_state    <= S0;

        end case;

    end process;

    -- combinational logic for outputs

    IR_write    <=  '1' when current_state = S0 else
                    '0';

    MAR_write   <=  '1' when current_state = S2A else
```

```
                    '0';

    DMD_read    <=  '1' when current_state = S3 else
                    '0';

    DMD_write   <=  '1' when current_state = S4B else
                    '0';

    RF_write    <=  '1' when current_state = S4A else
                    '0';

    PC_write    <=  '1' when current_state = S2E
                            or current_state = S2F
                            or current_state = S4A
                            or current_state = S4B
                            or current_state = S4C else
                    '0';

    HILO_write    <=  '1' when current_state = S2E
                            or current_state = S2F else
                    '0';

end Behavioral;
```

2.5 Τεχνική περιγραφή των μνημών

Μνήμη εντολών

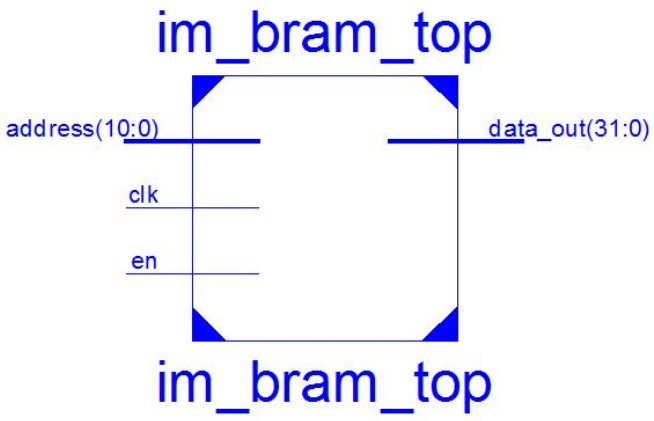
Γενική περιγραφή

Η μνήμη εντολών είναι μια ROM η οποία έχει αποθηκευμένες όλες τις εντολές του προγράμματος που θέλουμε να εκτελέσει ο επεξεργαστής μας. Οι εντολές είναι μεγέθους 32 bits συνεπώς η μνήμη εντολών μας επιτρέπει την αποθήκευση ενός πλήθούς εντολών των 32 bits. Ο καλύτερος τρόπος για να αποθηκεύσουμε τις εντολές αυτές στο FPGA είναι με χρήση BRAMs. Επειδή το μέγιστο μέγεθος μιας BRAM είναι 16 (+ 2 parity) ή 32 (+ 4 parity) Kbits αυτό σημαίνει οτι μπορούμε να αποθηκεύσουμε το μέγιστο 512 ή 1024 εντολές. Για να αποφύγουμε αυτόν τον περιορισμό χρησιμοποιούμε μια δομή με 4 BRAMs των 16 Kbits για συνολική χωρητικότητα 64KBits. Αυτό μας επιτρέπει να αποθηκεύσουμε το μέγιστο 2048 εντολές που είναι αρκετά καλύτερο και επιτρέπει τη φόρτωση ενός μεγάλου πλήθούς προγραμμάτων. Η επιλογή της BRAM από την οποία θα φορτωθεί η εντολή γίνεται μέσω των 2 MSBs της διεύθυνσης των 11 Bits.

Επειδή η χειροκίνητη εισαγωγή των εντολών είναι αρκετά επίπονη, αναπτύξαμε εργαλείο software και Makefile για το εργαλείο make το οποίο τοποθετεί δημιουργεί τις εντολές (machine code) από το πρόγραμμα μας σε assembly και τις τοποθετεί μέσα στα αρχεία που κάνουν instantiation των BRAMs. Παράλληλα φροντίζει για την αυτόματη αφαίρεση των NOP εντολών που παράγουν οι περισσότεροι assemblers για τις MIPS αρχιτεκτονικές λόγω της (στάνταρ) ύπαρξης branch delay slot στις περισσότερες αυτές αρχιτετκτονικές. Η μετατροπή της assembly σε machine code έγινε χρησιμοποιόντας τα GNU Binutils (as, ld) μεταγλωτισμένα για την αρχιτεκτονική MIPS (cross-compiled).

Όλοκληρο το toolchain καθώς και τα προγράμματα δοκιμής μας βρίσκονται στο φάκελο software ενώ ολόκληρη αυτή η διαδικασία μπορεί να εκκινηθεί απλά γράφωντας “make all” στην κονσόλα του Unix λειτουργικού μας.

Block διάγραμμα



VHDL κώδικας

```
library ieee;
use ieee.std_logic_1164.all;
--- 2K X 32 RAM (serial concatenation of four 512x32 block RAMs)
entity im_bram_top is
    port(   clk         : in  std_logic;
            en          : in  std_logic;
            address     : in  std_logic_vector(10 downto 0);
            data_out    : out std_logic_vector(31 downto 0));
end im_bram_top;

architecture Structural of im_bram_top is

    component im_bram_512x32_0 is
        port(   clk : in  std_logic;
                we  : in  std_logic;
                en  : in  std_logic;
                ssr : in  std_logic;
                a   : in  std_logic_vector(8 downto 0);
                di  : in  std_logic_vector(31 downto 0);
                do  : out std_logic_vector(31 downto 0);
                dop : out std_logic_vector(3 downto 0));
    end component;

    component im_bram_512x32_1 is
        port(   clk : in  std_logic;
                we  : in  std_logic;
                en  : in  std_logic;
                ssr : in  std_logic;
                a   : in  std_logic_vector(8 downto 0);
                di  : in  std_logic_vector(31 downto 0);
                do  : out std_logic_vector(31 downto 0);
                dop : out std_logic_vector(3 downto 0));
    end component;

    component im_bram_512x32_2 is
        port(   clk : in  std_logic;
                we  : in  std_logic;
                en  : in  std_logic;
                ssr : in  std_logic;
                a   : in  std_logic_vector(8 downto 0);
                di  : in  std_logic_vector(31 downto 0);
                do  : out std_logic_vector(31 downto 0);
                dop : out std_logic_vector(3 downto 0));
    end component;

    component im_bram_512x32_3 is
        port(   clk : in  std_logic;
                we  : in  std_logic;
                en  : in  std_logic;
                ssr : in  std_logic;
                a   : in  std_logic_vector(8 downto 0);
                di  : in  std_logic_vector(31 downto 0);
                do  : out std_logic_vector(31 downto 0);
                dop : out std_logic_vector(3 downto 0));
    end component;

    type do_array_type is array (natural range<>) of std_logic_vector(31 downto
0);

    signal do_internal  : do_array_type(0 to 3);
    signal sl           : std_logic_vector(3 downto 0);
```

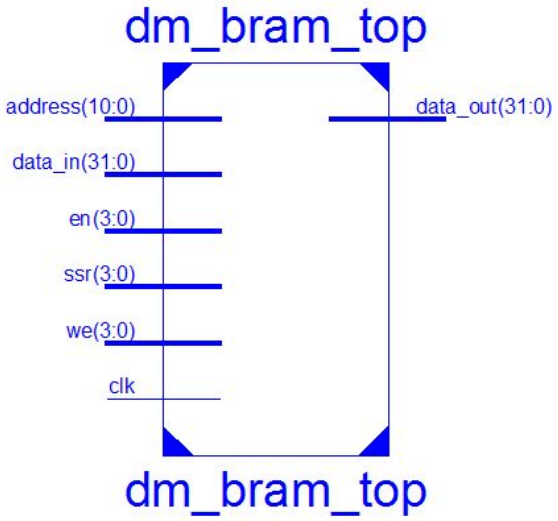
```
begin
    -- This module uses 4 512x32 block RAMs
    IM_0 : im_bram_512x32_0
    port map (
                clk => clk,
                we  => '0',
                en  => en,
                ssr => sl(0),
                a   => address (8 downto 0),
                di  => (others => '0'),
                do  => do_internal(0),
                dop => open);

    IM_1 : im_bram_512x32_1
    port map (
                clk => clk,
                we  => '0',
                en  => en,
                ssr => sl(1),
                a   => address (8 downto 0),
                di  => (others => '0'),
                do  => do_internal(1),
                dop => open);

    IM_2 : im_bram_512x32_2
    port map (
                clk => clk,
                we  => '0',
                en  => en,
                ssr => sl(2),
                a   => address (8 downto 0),
                di  => (others => '0'),
                do  => do_internal(2),
                dop => open);

    IM_3 : im_bram_512x32_3
    port map (
                clk => clk,
                we  => '0',
                en  => en,
                ssr => sl(3),
                a   => address (8 downto 0),
                di  => (others => '0'),
                do  => do_internal(3),
                dop => open);

    process (address)
    begin
        case  address (10 downto 9) is
            when "00"   => sl <= "1110";
            when "01"   => sl <= "1101";
            when "10"   => sl <= "1011";
            when "11"   => sl <= "0111";
            when others => sl <= "1111";
        end case;
    end process;

    data_out <= do_internal(0) or do_internal(1) or do_internal(2) or
do_internal(3);

end Structural;
```

Μνήμη δεδομένων

Γενική περιγραφή

Η μνήμη δεδομένων είναι μια RAM η οποία αποτελεί τον βασικό αποθηκευτικό χώρος για τα δεδομένα του προγράμματος που θέλουμε να εκτελέσει ο επεξεργαστής μας. Οι λέξεις είναι μεγέθους 32 bits συνεπώς η μνήμη δεδομένων μας επιτρέπει την φόρτωση ή αποθήκευση ενός πλήθούς λέξεων των 32 bits. Ο καλύτερος τρόπος για να αποθηκεύσουμε τις λέξεις αυτές στο FPGA είναι με χρήση BRAMs. Επειδή το μέγιστο μέγεθος μιας BRAM είναι 16 (+ 2 parity) ή 32 (+ 4 parity) Kbits αυτό σημαίνει οτι μπορούμε να αποθηκεύσουμε το μέγιστο 512 ή 1024 λέξεις. Για να αποφύγουμε αυτόν τον περιορισμό χρησιμοποιούμε μια δομή με 4 BRAMs των 16 Kbits για συνολική χωρητικότητα 64KBits. Η διασύνδεση τους γίνεται παράλληλα και κάθε ένα από τα 4 bytes της λέξης φορτώνεται ή αποθηκεύεται σε μια από τις 4 BRAMs ανάλογα με τα 2 LSBs της διεύθυνσης που ζητάμε. Αυτό μας επιτρέπει να αποθηκεύσουμε το μέγιστο 2048 λέξεις που είναι αρκετά καλύτερο και μας δίνει μεγαλύτερη ελευθερία.

Block διάγραμμα



VHDL κώδικας

```
library ieee;
use ieee.std_logic_1164.all;

--- 2K X 32 RAM (parallel concatenation of four 2Kx8 block RAMs)

entity dm_bram_top is
    port(   clk         : in  std_logic;
            en          : in  std_logic_vector(3 downto 0);
            we          : in  std_logic_vector(3 downto 0);
            ssr         : in  std_logic_vector(3 downto 0);
            address     : in  std_logic_vector(10 downto 0);
            data_in     : in  std_logic_vector(31 downto 0);
            data_out    : out std_logic_vector(31 downto 0));
end dm_bram_top;
```

```
architecture Structural of dm_bram_top is

    component dm_bram_2Kx8 is
        port(   clk : in  std_logic;
                we  : in  std_logic;
                en  : in  std_logic;
                ssr : in  std_logic;
                dop : out std_logic_vector(0 downto 0);
                a   : in  std_logic_vector(10 downto 0);
                di  : in  std_logic_vector (7  downto 0);
                do  : out std_logic_vector(7 downto 0));
    end component;

begin

    GenDM : for I in 0 to 3 generate

            DM : dm_bram_2Kx8
            port map(
                        clk => clk,
                        we  => we(I),
                        en  => en(I),
                        ssr => ssr(I),
                        a   => address,
                        di  => data_in(((8*I)+7) downto (8*I)),
                        do  => data_out(((8*I)+7) downto (8*I)),
                        dop => open);

    end generate GenDM;

end Structural;
```

2.6 Τεχνική περιγραφή όλου του επεξεργαστή (processor)

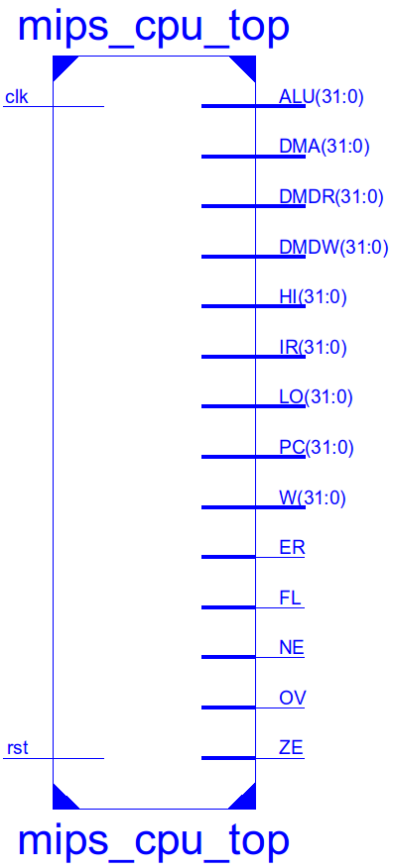
Γενική περιγραφή

Η μονάδα της διόδου δεδομένων περιλαμβάνει όλες τις μονάδες που αναλύσαμε στις προηγούμενες ενότητες. Πιο συγκεκριμένα περιλαμβάνει τις δύο μνήμες εντολών και δεδομένων, τη μονάδα ελέγχου (ακολουθιακό και συνδυαστικό κομμάτι) και τη δίοδο δεδομένων. Η διασύνδεση τους γίνεται με περιγραφή δομής και ακολουθεί κατά ένα μεγάλο μέρος τη διασύνδεση που προτείνεται στις σημειώσεις του μαθήματος.

Ολόκληρος ο επεξεργαστής κανονικά θα έπρεπε να έχει εισόδους μόνο τα σήματα ρολογιού (clk) και αρχικοποίησης (rst) και καθόλου εξόδους μιας και οι μνήμες εντολών και δεδομένων περιλαμβάνονται μέσα του. Αυτό όμως για λόγους αποσφαλμάτωσης δε συμβαίνει και έχουμε σαν έξοδο αρκετές αρτηρίες (32bit buses) και σήματα που θα θέλαμε γρήγορα να παρατηρήσουμε κατά τη διάρκεια της προσομοίωσης χωρίς κόπο. Αυτές οι debug αρτηρίες και σήματα εξόδου είναι οι εξής:

Όνομα	Περιγραφή
IR	Εντολή που εκτελείται
PC	Μετρητής προγράμματος
DMADDR	Διεύθυνση μνήμης δεδομένων
DMWE	Σήμα ενεργοποίησης εγγραφής (4 bit για κάθε ένα Byte της λέξης δεδμένων)
DMREAD	Δεδομένα ανάγνωσης από τη μνήμη δεδομένων
DMWRITE	Δεδομένα εγγραφής προς τη μνήμη δεδομένων
DMERROR	Αν η πρόσβαση από ή πρός τη μνήμη δεδομένων ήταν μη ευθυγραμμισμενη
RFWRITE	Δεδομένα εγγραφής προς το αρχείο καταχωρητών
ALU	Αποτέλεσμα πράξης της ALU
HI	Αποτέλεσμα πολλαπλασιαστή (32 MSBs)
LO	Αποτέλεσμα πολλαπλασιαστή (32 LSBs)
ZERO	Αν το αποτέλεσμα της πράξης της ALU είναι μηδέν
NEGATIVE	Αν το αποτέλεσμα της πράξης της ALU είναι αρνητικό
OVERFLOW	Αν η πράξη της ALU οδήγησε σε υπερχείλιση
BISTSTART	Αν η ενσωματωμένη αυτοδοκιμή του πολλαπλασιαστή ξεκίνησε
BISTDONE	Αν η ενσωματωμένη αυτοδοκιμή του πολλαπλασιαστή ολοκληρώθηκε
BIST FAIL	Αν η ενσωματωμένη αυτοδοκιμή του πολλαπλασιαστή ανίχνευσε λάθος

Block διάγραμμα



VHDL κώδικας

```
library ieee;
use ieee.std_logic_1164.all;

entity mips_cpu_top is
    generic(mult_pipe    : boolean := true);
    port (  clk          : in  std_logic;
            rst          : in  std_logic;
            IR           : out std_logic_vector(31 downto 0);
            PC           : out std_logic_vector(31 downto 0);
            DMA          : out std_logic_vector(31 downto 0);
            DMDR         : out std_logic_vector(31 downto 0);
            DMDW         : out std_logic_vector(31 downto 0);
            W            : out std_logic_vector(31 downto 0);
            ALU          : out std_logic_vector(31 downto 0);
            HI           : out std_logic_vector(31 downto 0);
            LO           : out std_logic_vector(31 downto 0);
            ZE           : out std_logic;
            NE           : out std_logic;
            OV           : out std_logic;
            FL           : out std_logic;
            ER           : out std_logic);
end mips_cpu_top;

architecture Structural of mips_cpu_top is
```

```
component im_bram_top is
    port(   clk          : in  std_logic;
            en           : in  std_logic;
            address      : in  std_logic_vector(10 downto 0);
            data_out     : out std_logic_vector(31 downto 0));
end component;

component dm_bram_top is
    port(   clk          : in  std_logic;
            en           : in  std_logic_vector(3 downto 0);
            we           : in  std_logic_vector(3 downto 0);
            ssr          : in  std_logic_vector(3 downto 0);
            address      : in  std_logic_vector(10 downto 0);
            data_in      : in  std_logic_vector(31 downto 0);
            data_out     : out std_logic_vector(31 downto 0));
end component;

component control_comb is
    generic(mult_pipe    : boolean := true);
    port(   clk          : in  std_logic;
            rst          : in  std_logic;
            OPCODE       : in  std_logic_vector(5 downto 0);
            FUNCT        : in  std_logic_vector(5 downto 0);
            RT           : in  std_logic_vector(4 downto 0);
            SorZ         : out std_logic;
            BorI         : out std_logic;
            ALUop        : out std_logic_vector(3 downto 0);
            sv           : out std_logic;
            MF           : out std_logic;
            MT           : out std_logic;
            HIorLO       : out std_logic;
            DMorALU      : out std_logic;
            DMWT         : out  std_logic_vector(2 downto 0);
            Link         : out std_logic;
            RorI         : out std_logic;
            BranchType   : out std_logic_vector(1 downto 0);
            NEorEQ       : out std_logic;
            RTZero       : out std_logic;
            Jump         : out std_logic;
            JumpPSD      : out std_logic;
            TestMult     : out std_logic);
end component;

component control_fsm is
    generic(mult_pipe    : boolean := true);
    port(   clk          : in  std_logic;
            rst          : in  std_logic;
            OPCODE       : in  std_logic_vector(5 downto 0);
            FUNCT        : in  std_logic_vector(5 downto 0);
            PC_write     : out std_logic;
            IR_write     : out std_logic;
            MAR_write    : out std_logic;
            DMD_read     : out std_logic;
            DMD_write    : out std_logic;
            RF_write     : out std_logic;
            HILO_write   : out std_logic);
end component;

component datapath_top is
    generic(mult_pipe       : boolean := true);
    port(   clk             : in  std_logic;
            rst             : in  std_logic;
```

```
            PC_write        : in  std_logic;
            RF_write        : in  std_logic;
            MAR_write       : in  std_logic;
            DMD_read        : in  std_logic;
            DMD_write       : in  std_logic;
            HILO_write      : in  std_logic;
            RorI            : in  std_logic;
            SorZ            : in  std_logic;
            BorI            : in  std_logic;
            sv              : in  std_logic;
            MF              : in  std_logic;
            MT              : in  std_logic;
            HIorLO          : in  std_logic;
            Jump            : in  std_logic;
            JumpPSD         : in  std_logic;
            BranchType      : in  std_logic_vector(1 downto 0);
            NEorEQ          : in  std_logic;
            RTZero          : in  std_logic;
            Link            : in  std_logic;
            DMorALU         : in  std_logic;
            DMWT            : in  std_logic_vector(2 downto 0);
            TestMult        : in  std_logic;
            ALUop           : in  std_logic_vector(3 downto 0);
            Bus_IRin        : in  std_logic_vector(31 downto 0);
            Bus_DMDin       : in std_logic_vector(31 downto 0);
            opcode          : out std_logic_vector(5 downto 0);
            funct           : out std_logic_vector(5 downto 0);
            rt              : out std_logic_vector(4 downto 0);
            Bus_FLAGSout    : out std_logic_vector(4 downto 0);
            Bus_PCout       : out std_logic_vector(31 downto 0);
            Bus_ALUout      : out std_logic_vector(31 downto 0);
            Bus_HIout       : out std_logic_vector(31 downto 0);
            Bus_LOout       : out std_logic_vector(31 downto 0);
            Bus_Wout        : out std_logic_vector(31 downto 0);
            Bus_DMWEout     : out std_logic_vector(3 downto 0);
            Bus_DMAout      : out std_logic_vector(31 downto 0);
            Bus_DMDout      : out std_logic_vector(31 downto 0));
    end component;

    signal PC_write     : std_logic;
    signal IR_write     : std_logic;
    signal RF_write     : std_logic;
    signal MAR_write    : std_logic;
    signal DMD_read     : std_logic;
    signal DMD_write    : std_logic;
    signal HILO_write   : std_logic;
    signal RorI         : std_logic;
    signal SorZ         : std_logic;
    signal BorI         : std_logic;
    signal sv           : std_logic;
    signal MF           : std_logic;
    signal MT           : std_logic;
    signal HIorLO       : std_logic;
    signal Jump         : std_logic;
    signal JumpPSD      : std_logic;
    signal BranchType   : std_logic_vector(1 downto 0);
    signal NEorEQ       : std_logic;
    signal RTZero       : std_logic;
    signal Link         : std_logic;
    signal DMorALU      : std_logic;
    signal DMWT         : std_logic_vector(2 downto 0);
    signal TestMult     : std_logic;
    signal ALUop        : std_logic_vector(3 downto 0);
```

```
    signal opcode        : std_logic_vector(5 downto 0);
    signal funct         : std_logic_vector(5 downto 0);
    signal rt            : std_logic_vector(4 downto 0);
    signal Bus_Flags     : std_logic_vector(4 downto 0);
    signal Bus_PCout     : std_logic_vector(31 downto 0);
    signal Bus_IRin      : std_logic_vector(31 downto 0);
    signal Bus_DMWE      : std_logic_vector(3 downto 0);
    signal Bus_DMA       : std_logic_vector(31 downto 0);
    signal Bus_DMDin     : std_logic_vector(31 downto 0);
    signal Bus_DMDout    : std_logic_vector(31 downto 0);
    signal dm_enable     : std_logic_vector(3 downto 0);

    begin
    dm_enable <=  (others => DMD_read or DMD_write);

    INSTMEM : im_bram_top
    port map(    clk          => clk,
                 en           => IR_write,
                 address      => Bus_PCout(12 downto 2),
                 data_out     => Bus_IRin);

    DATAMEM : dm_bram_top
    port map(    clk          => clk,
                 en           => dm_enable,
                 we           => Bus_DMWE,
                 ssr          => "0000",
                 address      => Bus_DMA(12 downto 2),
                 data_in      => Bus_DMDin,
                 data_out     => Bus_DMDout);

    CONTROLCOMB : control_comb
    generic map(mult_pipe    => mult_pipe)
    port map(    clk          => clk,
                 rst          => rst,
                 OPCODE       => opcode,
                 FUNCT        => funct,
                 RT           => rt,
                 SorZ         => SorZ,
                 BorI         => BorI,
                 ALUop        => ALUop,
                 sv           => sv,
                 MF           => MF,
                 MT           => MT,
                 HIorLO       => HIorLO,
                 DMorALU      => DMorALU,
                 DMWT         => DMWT,
                 Link         => Link,
                 RorI         => RorI,
                 BranchType   => BranchType,
                 NEorEQ       => NEorEQ,
                 RTZero       => RTZero,
                 Jump         => Jump,
                 JumpPSD      => JumpPSD,
                 TestMult     => TestMult);

    CONTROLFSM : control_fsm
    generic map(mult_pipe    => mult_pipe)
    port map(    clk          => clk,
                 rst          => rst,
                 OPCODE       => opcode,
                 FUNCT        => funct,
                 PC_write     => PC_write,
                 IR_write     => IR_write,
```

```
                MAR_write   => MAR_write,
                DMD_read    => DMD_read,
                DMD_write   => DMD_write,
                RF_write    => RF_write,
                HILO_write  => HILO_write);

    DATAPATH : datapath_top
    generic map(mult_pipe       => mult_pipe)
    port map(   clk             => clk,
                rst             => rst,
                PC_write        => PC_write,
                RF_write        => RF_write,
                MAR_write       => MAR_write,
                DMD_read        => DMD_read,
                DMD_write       => DMD_write,
                HILO_write      => HILO_write,
                RorI            => RorI,
                SorZ            => SorZ,
                BorI            => BorI,
                sv              => sv,
                MF              => MF,
                MT              => MT,
                HIorLO          => HIorLO,
                Jump            => Jump,
                JumpPSD         => JumpPSD,
                BranchType      => BranchType,
                NEorEQ          => NEorEQ,
                RTZero          => RTZero,
                Link            => Link,
                DMorALU         => DMorALU,
                DMWT            => DMWT,
                ALUop           => ALUop,
                TestMult        => TestMult,
                Bus_IRin        => Bus_IRin,
                Bus_DMDin       => Bus_DMDout,
                opcode          => opcode,
                funct           => funct,
                rt              => rt,
                Bus_PCout       => Bus_PCout,
                Bus_ALUout      => ALU,
                Bus_HIout       => HI,
                Bus_LOout       => LO,
                Bus_FLAGSout    => Bus_Flags,
                Bus_Wout        => W,
                Bus_DMWEout     => Bus_DMWE,
                Bus_DMAout      => Bus_DMA,
                Bus_DMDout      => Bus_DMDin);

    IR      <= Bus_IRin;
    PC      <= Bus_PCout;
    DMA     <= Bus_DMA;
    DMDR    <= Bus_DMDout;
    DMDW    <= Bus_DMDin;
    ZE      <= Bus_Flags(0);
    NE      <= Bus_Flags(1);
    OV      <= Bus_Flags(2);
    FL      <= Bus_Flags(3);
    ER      <= Bus_Flags(4);

end Structural;
```

3. Προσομοίωση

3.1 Μεθοδολογία προσομοίωσης (Behavioral & PAR)

Για την επαλήθευση της ορθής λειτουργίας όλων τον μονάδων αλλά και του επεξεργαστή συνολικά δημιουργήσαμε test-benches τα οποία και φορτώσαμε στον προσομοιωτή Isim που είναι ενσωματωμένος στο πακέτο ISE 14.5 της Xilinx.

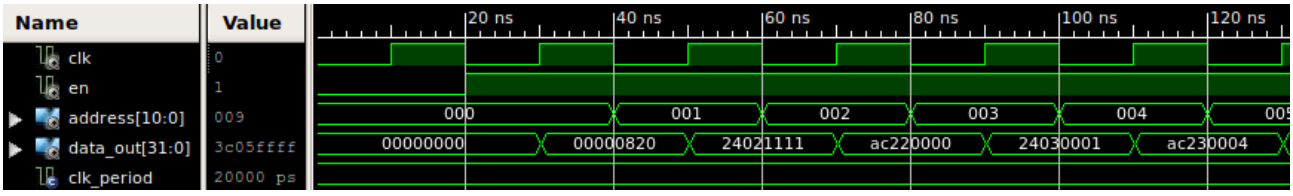
Μέσα στο φάκελο ./src/sim/ υπάρχουν όλα τα test-bench που δημιουργήσαμε. Όλες ανεξαιρέτως οι μονάδες έχουν το δικό τους test-bench και γενικότερα ο φάκελος αυτός ακολουθεί πιστά τη δομή του φακέλου ./src/rtl/ που περιέχει την RTL περιγραφή του επεξεργαστή ώστε να μπορεί κάποιος να βρεί το αντίστοιχο test-bench εύκολα και γρήγορα.

Αρχικά προσομοιώσαμε κάθε μονάδα ξεχωριστά ακριβώς μετά απο την ολοκλήρωση της συγγραφής της RTL περιγραφής της. Ανάλογα την πολυπλοκότητα της κάθε μονάδας το αντίστοιχο test-bench είναι μικρό και απλό ή μεγαλύτερο και πιο εξαντλητικό ώστε να καλύψει όλα τα δυνατά σενάρια λειτουργίας της μονάδας. Για παράδειγμα το test-bench μιας απλής μονάδας που περιέχει έναν πολυπλέκτη 4 προς 1 είναι πολύ απλό και δοκιμάζει μόνο 4 διαφορετικές εισόδους ώστε να επαληθευτεί οτι ανάλογα τον επιλογέα του πολυπλέκτη βγαίνει στην έξοδο η σωστή είσοδος. Εν αντιθέση μια πολύπλοκη μονάδα όπως η ALU ή το Register File έχει πολύ μεγαλύτερες απαιτήσεις για την επαλήθευση της ορθής λειτουργίας της με όλα τα δυνατά σενάρια εισόδων.

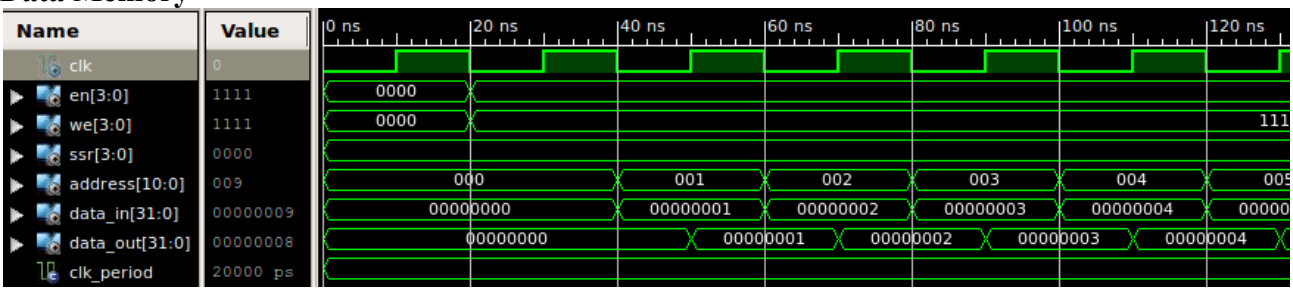
Στο τέλος και αφού ολοκληρώσαμε ολόκληρο το σχέδιο του επεξεργαστή δοκιμάσαμε να εκτελέσουμε timing (par) προσομοιώσεις κυρίως για ολόκληρο τον επεξεργαστή. Η επαλήθευση ορθής λειτουργίας ακόμα και με αυτόν τον τρόπο προσομοίωσης έγινε άμεσα με μοναδική αλλαγή σε μια γραμμή κώδικα. Αυτό δεν οφείλεται σε απλή τύχη αλλά σίγουρα είναι αποτέλεσμα των σωστών πρακτικών που ακολουθήσαμε κατα τη δημιουργία των RTL περιγραφών της κάθε μονάδας που υλοποιήσαμε.

3.2 Προσομοίωση των επιμέρους πιο σημαντικών entities

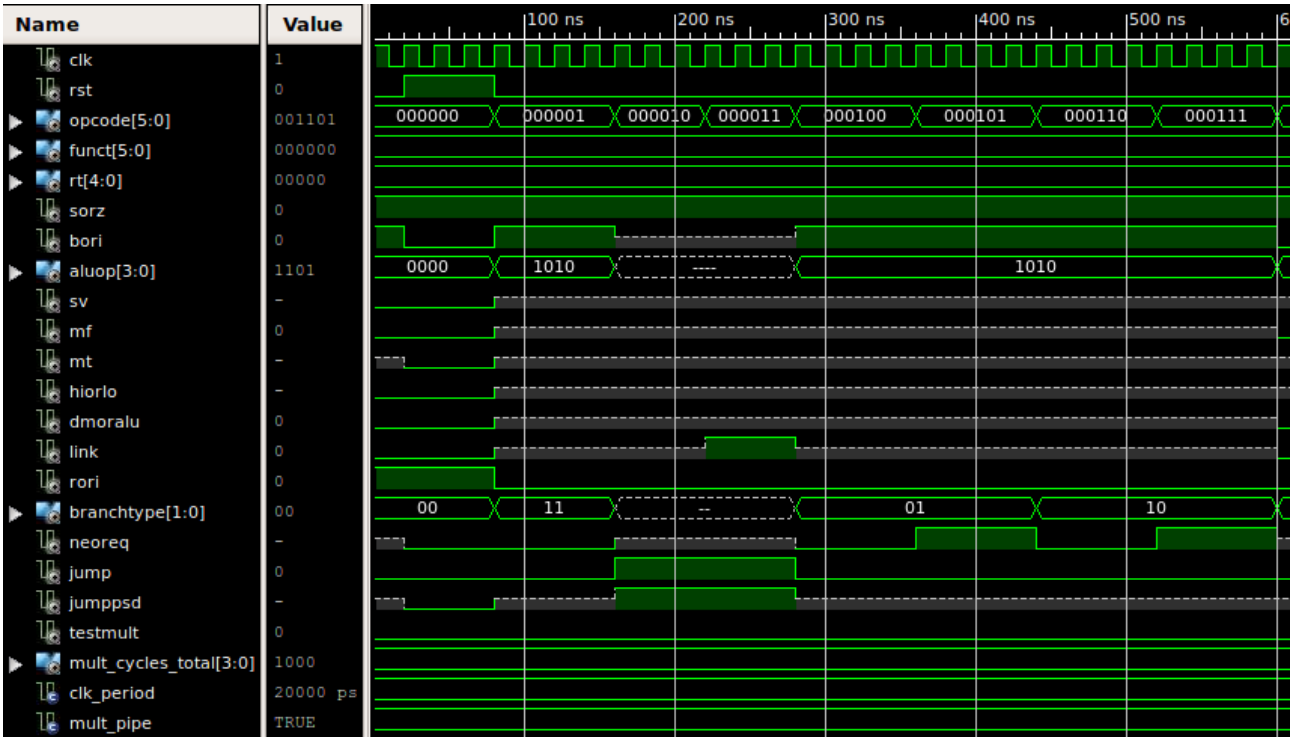
Instruction Memory



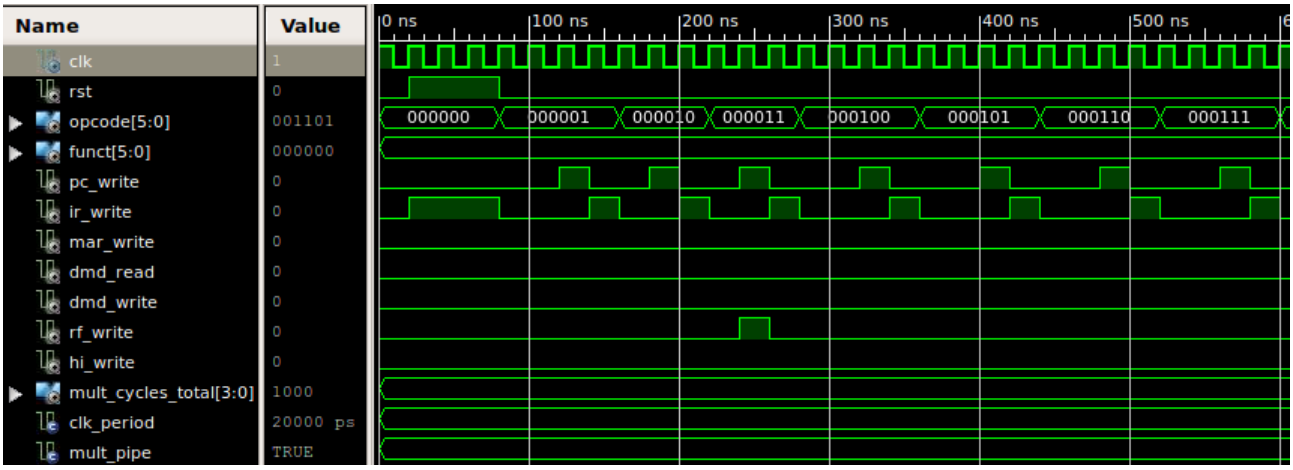
Data Memory



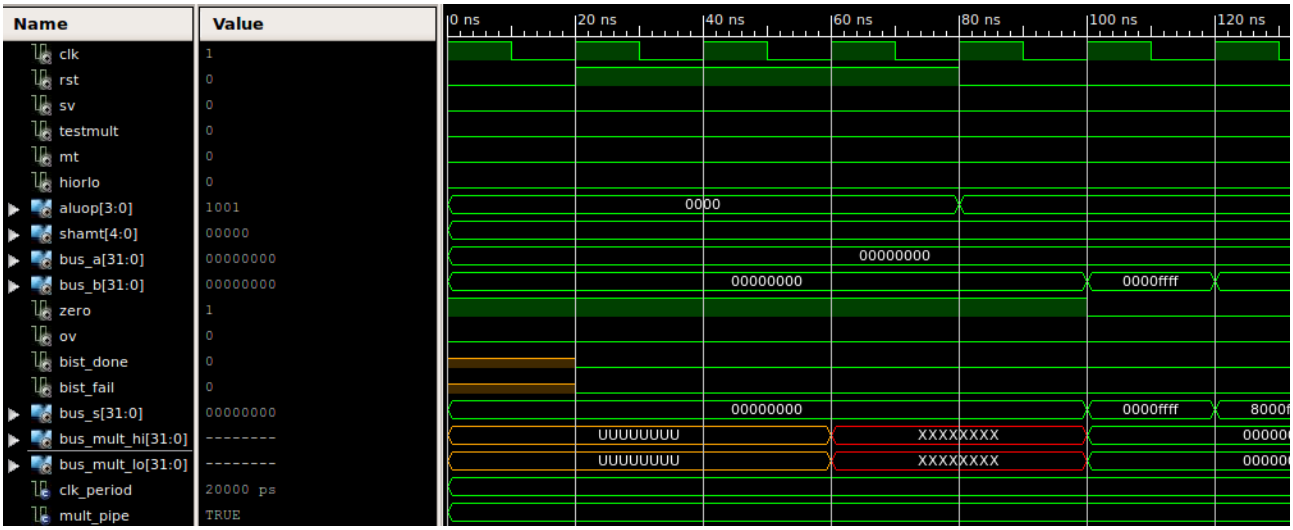
Control Combinational



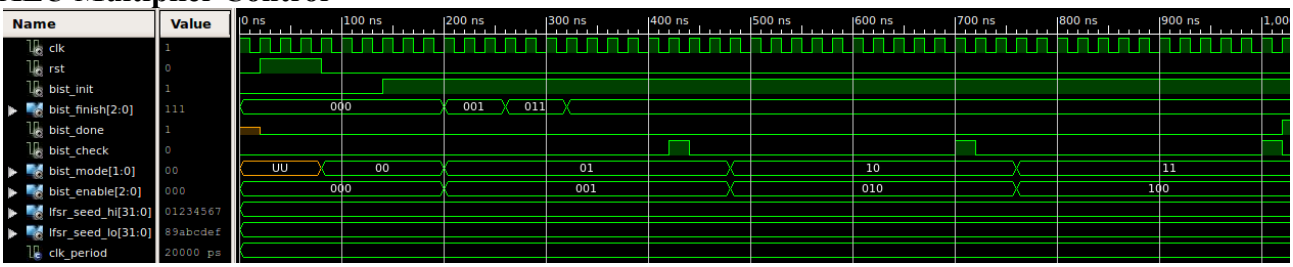
Control FSM



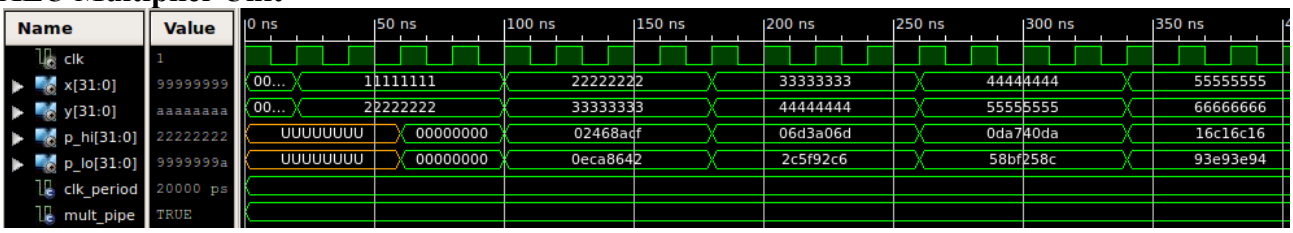
ALU



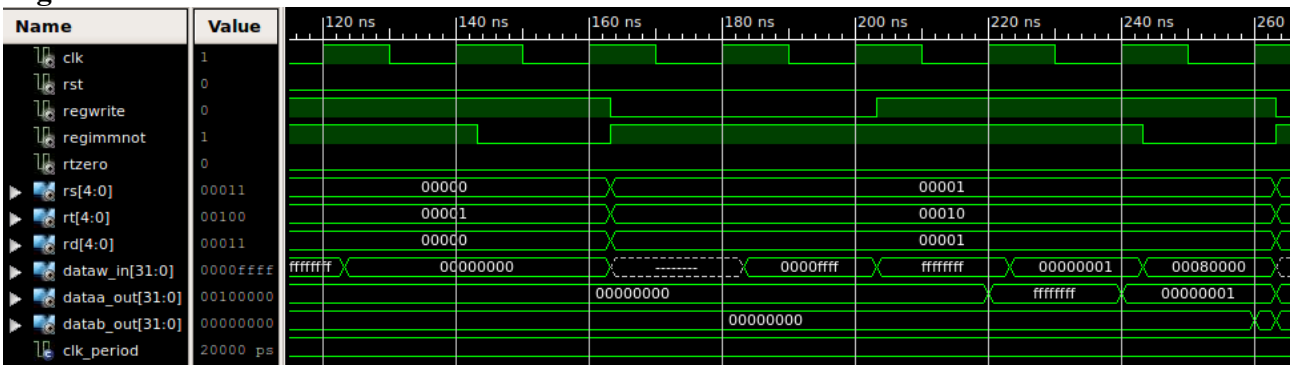
ALU Multiplier Control



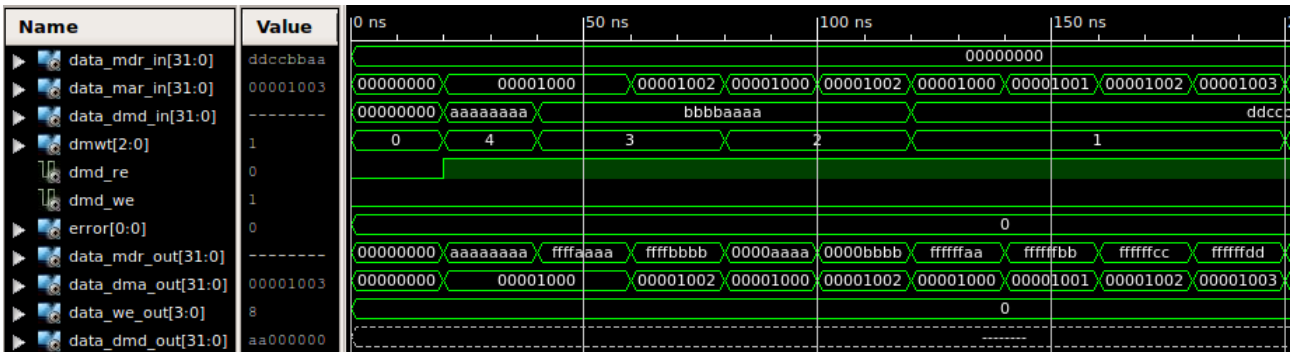
ALU Multiplier Unit



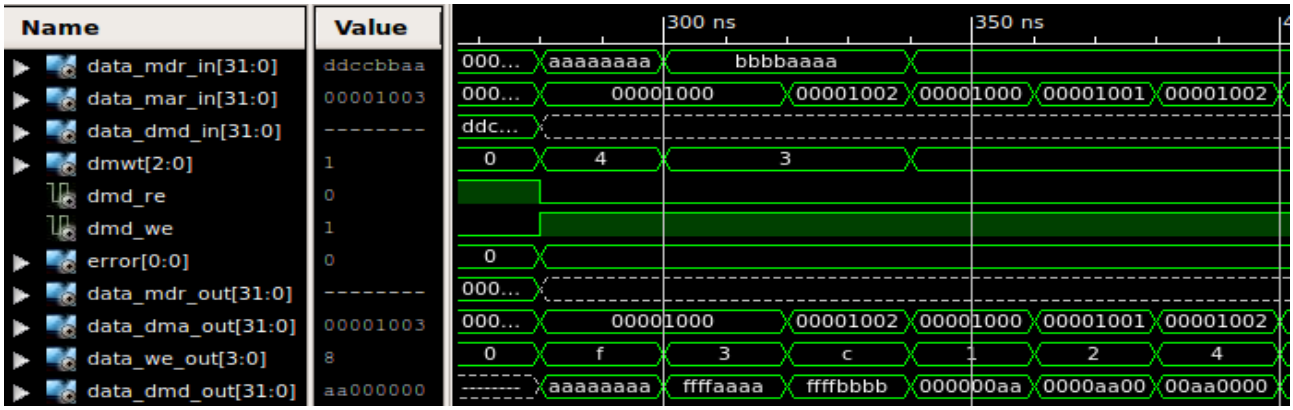
Register File



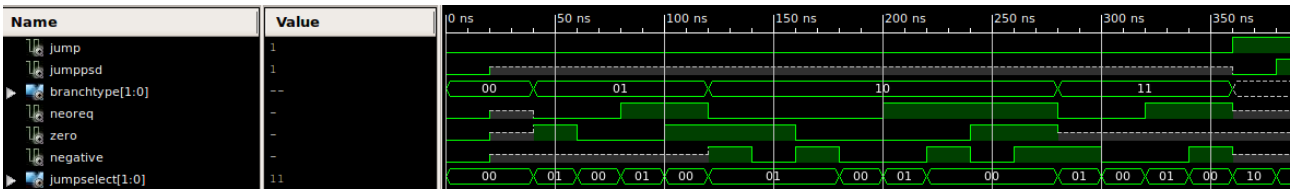
DM Control Read



DM Control Write



NPC Selector



3.3 Προσομοίωση ολόκληρου του επεξεργαστή

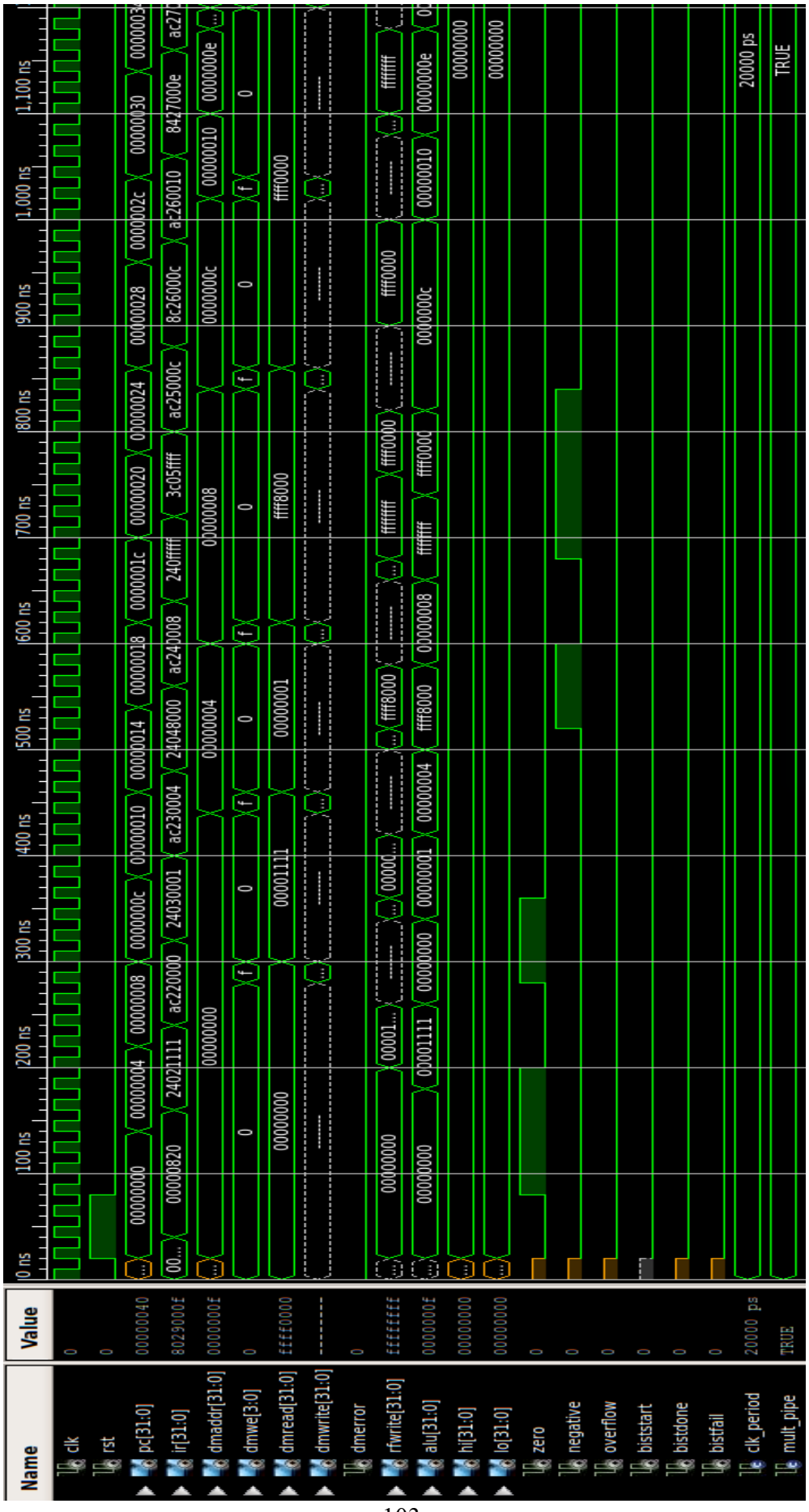
Στην ενότητα αυτή παρουσιάζομε την προσομοίωση ολόκληρου του επεξεργαστή χρησιμοποιώντας μερικά από τα προγράμματα assembly που αναπτύξαμε.

full.s

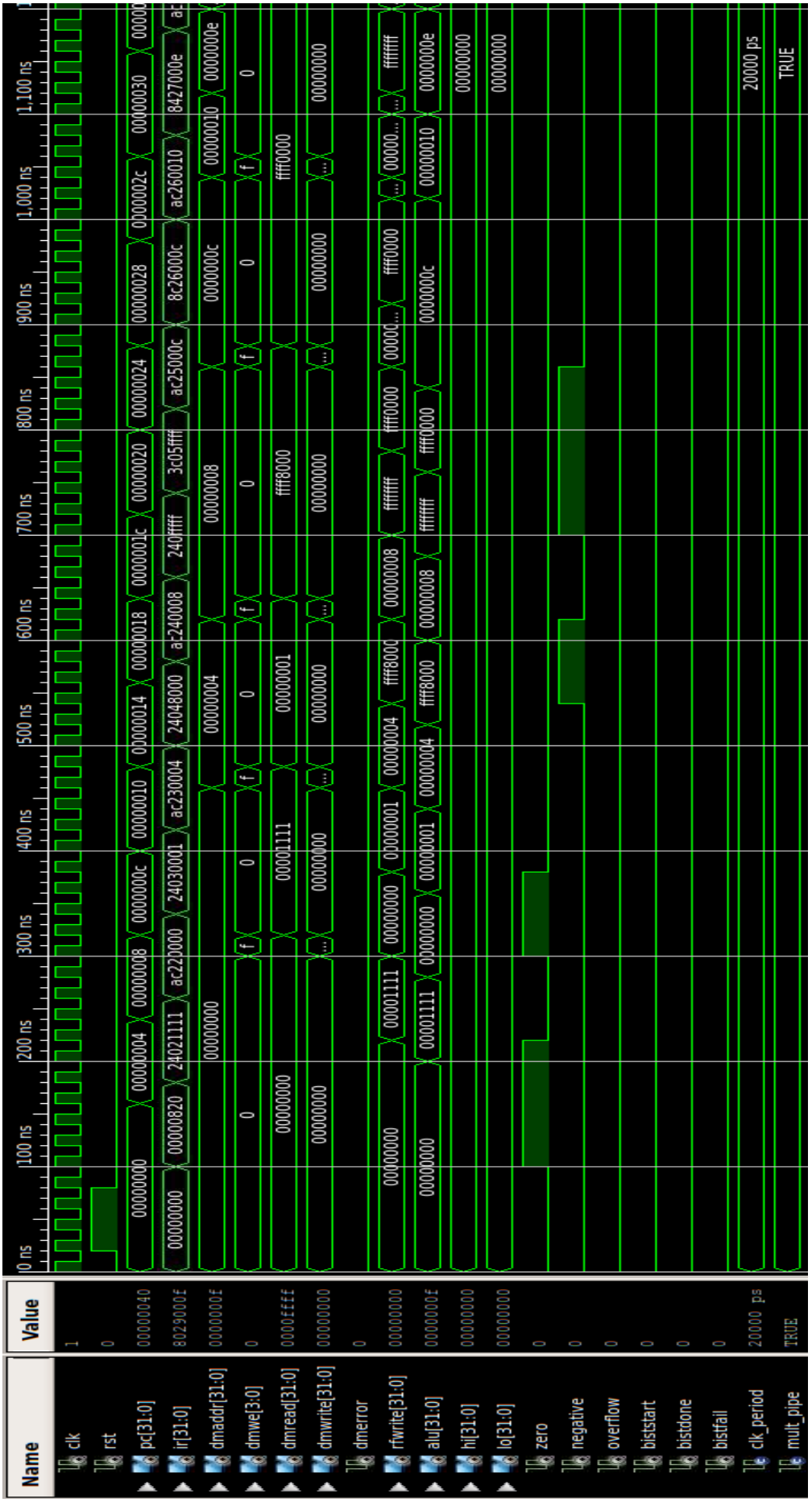
Το πρόγραμμα αυτό εκτελεί όλες τις εντολές που υποστηρίζει ο επεξεργαστής και για κάθε μια αποθηκεύει το αποτέλεσμα της σε συγκεκριμένη θέση μνήμης. Στο τέλος και αφού ολοκληρωθούν όλες οι κανονικές εντολές ξεκινάει, με την εντολή TEST, τις μεθόδους ενσωματωμένης αυτοδοκιμής του πολλαπλασιαστή (BIST). Αυτές εκτελούνται με τη σειρά η καθεμία (LFSR, Deterministic Counter, ATPG) και αν ανιχνευτεί το οποιοδήποτε πρόβλημα και άρα λάθος αποτέλεσμα, ενεργοποιείται το σήμα BISTFAIL. Προφανώς στο ελεγχόμενο και ιδανικό περιβάλλον της προσομοίωσης δε πρόκειται ποτέ να παρατηρήσουμε κάποιο πρόβλημα στο κύκλωμα όπως είναι π.χ ένα stack at 0 or 1 bit το οποίο θα οδηγούσε γρήγορα σε λάθος αποτελέσματα.

Σε κάθε περίπτωση η έναρξη και η λήξη της ενσωματωμένης αυτοδοκιμής είναι ορατή στο χρήστη από τα σήματα BISTSTART και BISTDONE αντίστοιχα.

Behavioral Simulation

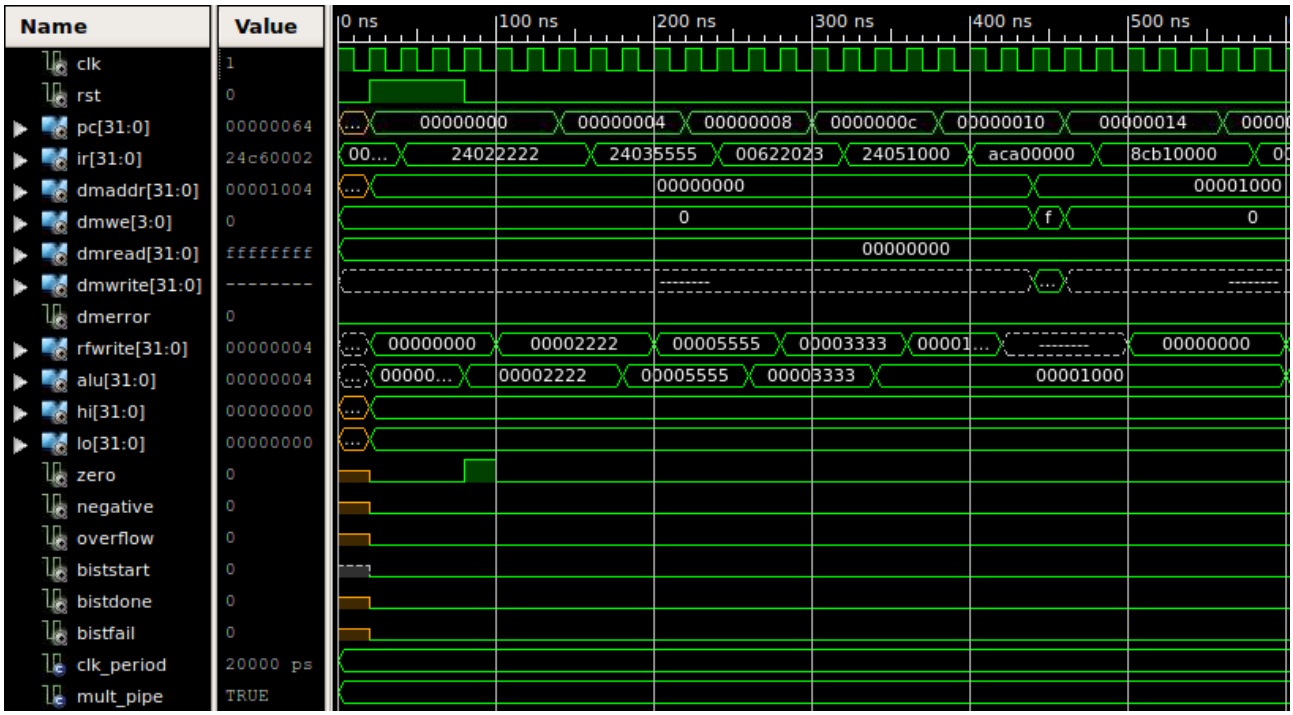


Par Simulation



simple.s

Το απλό αυτό πρόγραμμα εκτελεί μερικές από τις εντολές που υποστηρίζονται και αναπτύχθηκε στα πολύ αρχικά στάδια ανάπτυξης ώστε να δοκιμαστεί ο επεξεργαστής με ένα μικρό υποσύνολο εντολών. Παρόλη την απλότητα του είναι αρκετά χρήσιμο για να παρατηρήσει ο χρήστης την ορθή αλλαγή ροής του προγράμματος μιας και στο δεύτερο κομμάτι του εκτελεί αρκετές εντολές branch και jump οι οποίες χρησιμοποιούνται για την εκτέλεση loops ή και function calls.

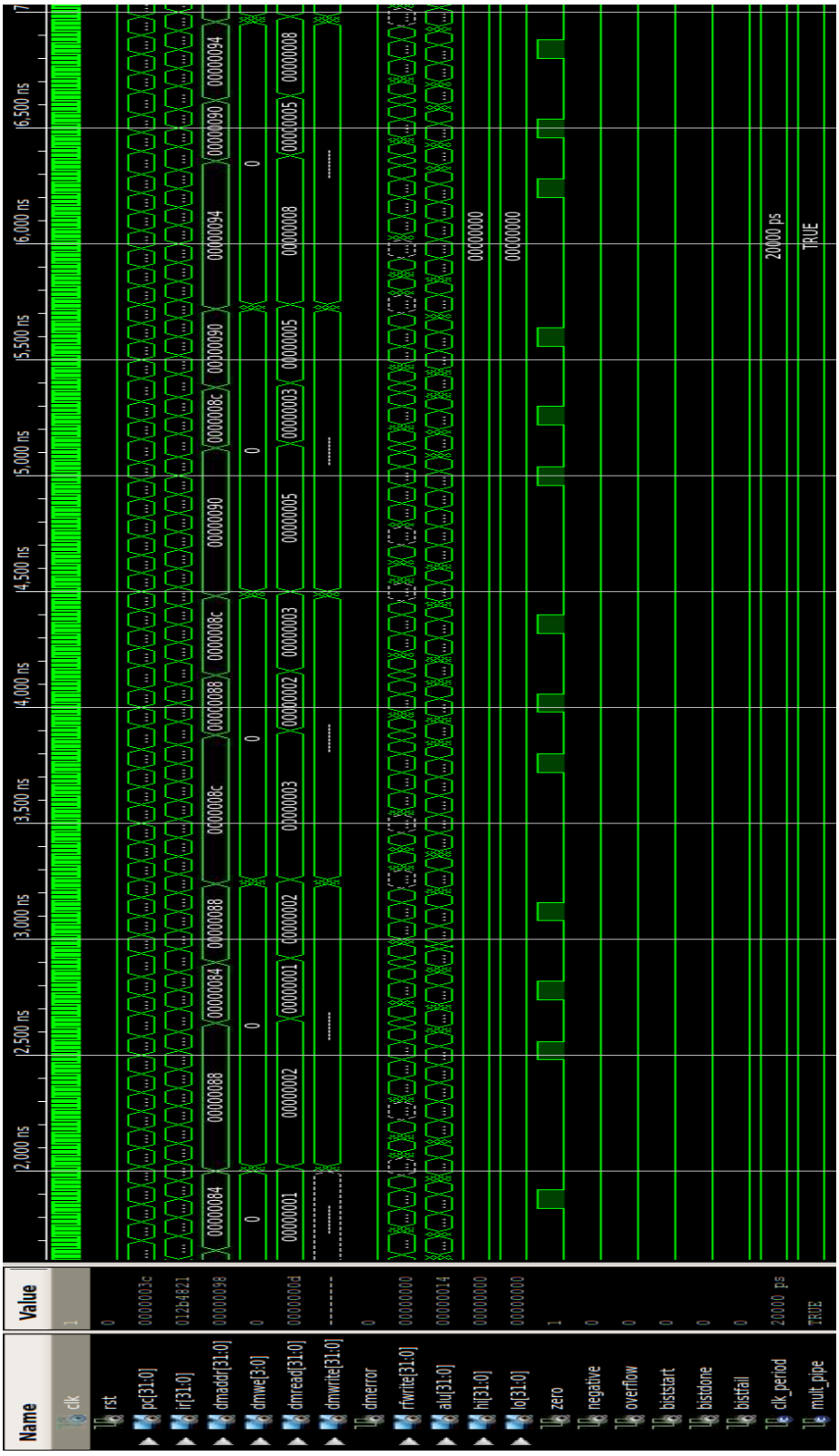


matrix.s

Το τελευταίο πρόγραμμα που παρουσιάζουμε στην ενότητα αυτή είναι ένα κλασικό πρόγραμμα πολλαπλασιασμού πινάκων. Υπολογίζει τον πίνακα P = A x B ο οποίος έχει τόσες γραμμές όσες ο πίνακας A και τόσες στήλες όσο ο πίνακας B. Από τις ιδιότητες του πολλαπλασιασμού πινάκων είναι απαραίτητο ο αριθμός των στηλών του A να είναι ίσος με τον αριθμό των γραμμών του B. Δηλαδή P(M x N) = A(M x R) X B(R x N).

fibonacci.s

Το συγκεκριμένο πρόγραμμα μας δώθηκε έτοιμο και όπως είναι προφανές εκτελεί την εύρεση των αριθμών fibonacci. Τερματίζει όταν υπολογίσει τους πρώτους 40 αριθμούς fibonacci.



4. Αποτελέσματα Υλοποίησης

Ακολουθούν τα στατιστικά στοιχεία που υπολογίζει το εργαλείο ISE 14.5 της Xilinx ύστερα από μόνο την επιτυχή σύνθεση του επεξεργαστή και ύστερα από την επιτυχή σύνθεση και μετέπειτα την "υλοποίηση" του επεξεργαστή πάνω στο επιλεγμένο FPGA της οικογένειας Spartan-3 της Xilinx (xc3s1000-5fg676). Στή δεύτερη περίπτωση η "υλοποίηση" περιλαμβάνει όλες τις υπο-διεργασίες όπως είναι η translate, ή map και η place and route. Η τελική μέγιστη συχνότητα του επεξεργαστή είναι τα 80 MHz με περίοδο 12.490 ns και κρίσιμο μονοπάτι που φαίνεται στην επόμενη ενότητα.

4.1 Στατιστικά στοιχεία

Στατιστικά σύνθεσης

HDL Synthesis Report:	
Macro Statistics	
# RAMs	2
32x32-bit dual-port RAM	2
# ROMs	4
128x32-bit ROM	2
4x2-bit ROM	1
4x3-bit ROM	1
# Multipliers	1
32x32-bit multiplier	1
# Adders/Subtractors	10
2-bit adder	1
3-bit adder	1
32-bit adder	2
33-bit adder	2
33-bit subtractor	2
7-bit adder	1
8-bit adder	1
# Counters	2
2-bit up counter	1
3-bit up counter	1
# Registers	117
1-bit register	85
2-bit register	3
3-bit register	3
32-bit register	16
4-bit register	1
5-bit register	2
64-bit register	5
7-bit register	1
8-bit register	1
# Comparators	1
64-bit comparator equal	1
# Multiplexers	15
1-bit 4-to-1 multiplexer	3
32-bit 4-to-1 multiplexer	12
# Logic shifters	3
32-bit shifter arithmetic right	1

32-bit shifter logical left	1
32-bit shifter logical right	1
# Decoders	1
1-of-4 decoder	1
# Xors	7
1-bit xor2	3
1-bit xor4	2
32-bit xor2	1
64-bit xor2	1

Device utilization summary:	
Selected Device:	3s1000fg676-5
Number of Slices:	1754 out of 7680 22%
Number of Slice Flip Flops:	1382 out of 15360 8%
Number of 4 input LUTs:	3432 out of 15360 22%
Number used as logic:	3114
Number used as Shift registers:	62
Number used as RAMs:	256
Number of IOs:	301
Number of bonded IOBs:	301 out of 391 76%
IOB Flip Flops:	1
Number of BRAMs:	10 out of 24 41%
Number of GCLKs:	1 out of 8 12%

Timing Summary:

Minimum period: 13.899ns (Maximum Frequency: 71.950MHz)
Minimum input arrival time before clock: 9.864ns
Maximum output required time after clock: 19.403ns
Maximum combinational path delay: No path found

Στατιστικά PAR

Device utilization summary:			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops:	1,382	15,360	8%
Number of 4 input LUTs:	3,368	15,360	21%
Number of occupied slices:	1,845	7,680	24%
Number of slices containing only related logic	1,845	1,845	100%
Number of containing unrelated logic	0	1,845	0%
Total Number of 4 input LUTs:	3,491	15,360	22%
Number used as logic	3,050		
Number used route-thru	123		
Number used for Dual Port Rams	256		
Number used as Shift Registers	62		
Number of bonded IOBs:	301	391	76%
IOB Flip Flops:	1		
Number of BRAM16s:	10	24	41%
Number of BUFGMUXs:	1	8	12%
Anerage Fanout Non-Clock Nets	3,27		

Timing summary:

Timing errors: 0 Score: 0 (Setup/Max: 0, Hold: 0)
Constraints cover 1836113 paths, 0 nets, and 13396 connections
Minimum period: 12.490ns{1} (Maximum frequency: 80.064MHz)

4.2 Προσδιορισμός του critical path

Το critical path όπως υπολογίστηκε από το εργαλείο static timing είναι το εξής:

Slack (setup path):	0.010ns (requirement - (data path - clock path skew + uncertainty))
Source:	DATAPATH/I/data_out_16 (FF)
Destination:	DATAPATH/ALUOUT/data_out_26 (FF)
Requirement:	12.500ns
Data Path Delay:	12.490ns (Levels of Logic = 5)
Clock Path Skew:	0.000ns
Source Clock:	clk_BUFGP rising at 0.000ns
Destination Clock:	clk_BUFGP rising at 12.500ns
Clock Uncertainty:	0.000ns

Maximum Data Path: DATAPATH/I/data_out_16 to DATAPATH/ALUOUT/data_out_26			
Location	**Delay type**	**Delay(ns)**	**Physical Resource Logical Resource(s)**
SLICE_X27Y33.XQ	Tcko	0.626	DATAPATH/I/data_out<16>
			DATAPATH/I/data_out_16
SLICE_X27Y19.G1	net (fanout=99)	2.987	DATAPATH/I/data_out<16>
SLICE_X27Y19.Y	Tilo	0.479	DATAPATH/ALU/Sh24320
			DATAPATH/ALUMUX/dataB_out<31>1
SLICE_X38Y16.G3	net (fanout=36)	1.506	DATAPATH/Bus_ALUMUXB<31>
SLICE_X38Y16.Y	Tilo	0.529	DATAPATH/ALU/Sh254
			DATAPATH/ALU/Sh2221
SLICE_X29Y15.G4	net (fanout=7)	1.104	DATAPATH/ALU/Sh222
SLICE_X29Y15.X	Tif5x	0.793	DATAPATH/ALU/Sh250
			DATAPATH/ALU/Sh250_F
			DATAPATH/ALU/Sh250
SLICE_X31Y13.F1	net (fanout=2)	0.799	DATAPATH/ALU/Sh250
SLICE_X31Y13.X	Tif5x	0.793	DATAPATH/ALU/Mmux_Bus_S136223
			DATAPATH/ALU/Mmux_Bus_S136223_G
			DATAPATH/ALU/Mmux_Bus_S136223
SLICE_X35Y12.G1	net (fanout=1)	0.893	DATAPATH/MAR/data_out<26>
SLICE_X35Y12.X	Tif5x	0.793	DATAPATH/ALU/Mmux_Bus_S136313_F
			DATAPATH/ALU/Mmux_Bus_S136223
			DATAPATH/ALU/Mmux_Bus_S136313
SLICE_X37Y25.BY	net (fanout=2)	0.961	ALU_26_OBUF
SLICE_X37Y25.CLK	Tdick	0.227	DATAPATH/ALUOUT/data_out<27>
			DATAPATH/ALUOUT/data_out_26
Total		**12.490ns**	**(4.240ns logic, 8.250ns route)**
			(33.9% logic, 66.1% route)

Μελέτη δυνατότητας πιθανής βελτιστοποίησης (area, speed)

Η βελτισοποίηση του επεξεργαστή μας μπορεί να έχει δύο διαφορετικούς στόχους οι οποίοι συνήθως (αλλά οχι πάντα) είναι αντίθετοι μεταξύ τους. Αυτοί οι δύο είνα η βελτιστοποίηση της επιφάνειας και η βελτιστοποίηση της ταχύτητας. Είναι πού λογικό να θέλουμε να βελτιώσουμε και τα δύο, όμως σε κάθε περίπτωση θα πρέπει να βάλουμε προτεραιότητα στον ένα από τους δύο στόχους ανάλογα με τις προδιαγραφές και το περιβάλλον στο οποίο θα λειτουργήσει τελικά ο επεξεργαστής μας. Από τη μεριά μας γνωρίζαμε εκ των προτέρων το FPGA στο οποίο στοχεύαμε να είναι λειτουργικό το σχέδιο. Αυτο είναι το Spartan 3 xc3s1000-5fg676 το οποίο είναι ένα σχετικά μεγάλο FPGA για τις ανάγκες της υλοποίησης ενός MIPS R2000 επεξεργαστή. Συνεπώς επειδή δεν υπήρχε σε καμία περίπτωση περιορισμός επιφάνειας, οι όποιες βελτιστοποιήσεις μας είχαν σα πρωταρχικό στόχο την ταχύτητα του επεξεργαστή.

Κατά τη σχεδίαση και την υλοποίηση έγινε προσπάθεια και για τη βελτιστοποίηση της επιφάνειας όμως αυτή πάντα ερχόταν σε δεύτερη μοίρα σε σχέση με την ταχύτητα. Παρόλα αυτά μπορούμε να πούμε οτι τα καταφέραμε αρκετά καλά και σε αυτόν τον τομέα καθώς το σχέδιο μας χρησιμοποιεί μόλις το 24% των slices που διαθέτει το FPGA κάτι που είναι αρκετά εντυπωσιακό, τόσο επειδή σε μόλις τόσα slices υλοποιείται ένας ολόκληρος MIPS R2000 επεξεργαστής με ένα ρεπερτόριο 48 εντολών, όσο και επειδή ο πρωταρχικός μας στόχος όπως αναφέραμε ήταν η ταχύτητα και όχι η επιφάνεια κάλυψης. Οι βελτιστοποιήσεις επιφάνειας είχαν έμφαση στην ελαχιστοποίηση της απαραίτητς λογικής όπου αυτό χρειαζόταν. Αυτό έγινε κυρίως στις FSM του σχεδίου όπου προσπαθήσαμε το σχέδιο μας να έχει τις ελάχιστες δυνατές καταστάσεις. Επίσης όλα τα σήματα ελέγχου μελετήθηκαν προσεχτικά και αξιολογήθηκε η αναγκαιότητα του καθενώς από αυτά. Όσα δεν ήταν απαραίτητα αφαιρέθηκαν η προστέθηκε λογική “don't care”.

Όσον αφορά τη βελτιστοποίηση της ταχύτητας, κάναμε σχεδόν ότι ήταν δυνατό για να επιτύχουμε αυτό το στόχο με βάση τους περιορισμούς που είχαμε και τις γνώσεις που διαθέτουμε πάνω στο αντικείμενο. Επίσης πρέπει να επαναλάβουμε οτι τις όποιες βελτιστοποιήσεις της ταχύτητας της εκτελέσαμε μόνο αφού είμασταν βέβαιοι πως το σχέδιο μέχρι εκείνο το σημειο ήταν λειτουργικό και παρήγαγε ορθά αποτελέσματα κατά τη σύνθεση τόσο σε επίπεδο συμπεριφοράς όσο και κατά την timing (par) μελέτη και προσομοίωση.

Η βελτιστοποίηση ταχύτητας έγινε μελετώντας κυρίως τις καθυστερήσεις διάδοσης (κρίσιμα μονοπάτια) μεταξύ των συνεδεμένων μονάδων αλλά και της κάθε μονάδας ξεχωριστά. Αφού μελετήσαμε την καθυστέρηση διάδοσης μεταξύ των καταχωρητών του κάθε σταδίου εκτέλεσης της εντολής, προσθέσαμε ενδιάμεσους προσωρινος καταχωρητές όπου αυτό ήταν απαραίτητο και φυσικά είχε νόημα, δίνωντας μας μικρότρα κρίσιμα μονοπάτια, μικρότερη ελάχιστη περίοδο και άρα μεγαλύτερη μέγιστη συχνότητα λειτουργίας. Ανδιαφησβήτιτα η αναφορά static timing που παράγει το εργαλείο ISE 14.5 της Xilinx μετά τη διαδικασία PAR ήταν ο μεγαλύτερος μας σύμμαχος στην επίτευξη αυτού του στόχου. Οι βελτιστοποιήσεις που τελικά κρατήσαμε στο τελικό πλήρως επιβεβαιωμένο σχέδιο είναι οι εξής:

Control Comb Registers

Προστέθηκαν registers σε όλες τις εξόδους της μονάδας control_comb εκτός από αυτές οι οποίες χρειάζονται σε μονάδες που λειτουργούν κατά τη διάρκεια της δεύτερης φάσης (ID) όπως είναι το αρχείο καταχωρητών και η μονάδα επέκτασης προσήμου/μηδενός (SorZ, RorI, RTZero).

Control FSM States

Όλες οι καταστάσεις της μονάδας control_fsm είναι προσεγμένες ώστε να υπάρχουν μόνο οι ελάχιστες δυνατές.

Pipelined Multiplier

Υπάρχε επιλογή για χρήση pipelined (default) ή κανονικού πολλαπλασιαστής. Ο pipelined πολλαπλασιαστής ολοκληρώνει την πράξη του πολλαπλασιασμού σε 4 κύκλους ρολογιού ενώ ο κανονικός σε 1. Η επιλογή του τύπου του πολλαπλασιαστή γίνεται από το χρήστη μέσω μιας generic παραμέτρου.

Register File using Distributed RAM

Το αρχείο καταχωρητών είναι υλοποιημένο με κατανεμημένη μνήμη αντί γα BRAM. Αυτό επιτρέπει στις εξόδους διαβάσματος να είναι ασύγχρονες

ALU Mult Multiplexer Register

Τοποθετήθηκε καταχωρητής στην έξοδο του πολυπλέκτη της μονάδας του πολλαπλασιαστή. Ο πολυπλέκτης αυτός χρησιμοποιείται για να επιλεγεί η είσοδος του πολλαπλασιαστή μεταξύ της κανονικής λειτουργίας (εντολή MULT) και της λειτουργίας ενσωματωμένης αυτοδοκιμής με τις τρείς διαφορετικούς μεθόδους (εντολή TEST). Η τοποθέτηση καταχωρητή σε αυτό το σημείο αυξάνει τον αριθμό κύκλων του πολλαπλασιασμού (όπως και ο pipelined πολλαπλασιαστής) όμως πιστεύουμε πως αυτό δεν είναι πρόβλημα σε μια εντολή / πράξη που θεωρείται γενικά «ακριβή».

NPC Selector Multiplexer

Δώθηκε προσοχή στην ελαχιστοποίηση των σημάτων ελέγχου που καταλήγουν σε αυτή τη μονάδα ώστε να μειωθεί το συνδυαστικό control. Λόγω των λιγότερων σημάτων η μονάδα αυτή έπρεπε να υλοποιηθεί με μεγαλύτερη δυσκολία και έξυπνο κώδικα περιγραφής RTL. Το αποτέλσμα είναι ο πολυπλέκτης (μια ακριβή μονάδα στα FPGA ειδικά αν είναι πολλών εισόδων) που περιλαμβάνει να έχει 4 εισόδους αντί για αρκετές περισσότερες (τουλάχιστον 7).

Τέλος να αναφέρουμε οτι θα θέλαμε αλλά δυστυχώς δε μας επιτράπηκε να κάνουμε το όλο σχέδιο pipelined ώστε η κάθε εντολή να μη περιμένει την εκτέλεση όλων των κύκλων της προηγούμενης. Το επόμενο στάδιο θα ήταν και η εκμετάλευση της περισευούμενης επιφάνειας με την επανάληψη μερικών μονάδων ιδιαίτερα αυτών της φάσης εκτέλεσης (execution) όπως είναι η ALU ώστε το σχέδιο μας να γίνει superscalar και να μπορεί να εκτελεί όπου αυτό είναι δυνατό πάνω από μια εντολές ταυτόγχρονα. Οι δύο αυτές τεχνικές εφαρμόζονται σε όλους τους σύγχρονους επεξεργαστές και ανεβάζουν κατακόρυφα το επίπεδο της απόδοσης (IPC) του επεξεργαστή στον οποίο υλοποιούνται. Μπορούν επίσης να βελτιώσουν την κατανάλωση του συστήματος καθώς σε αυτές τις περιπτώσεις το σύστημα ολοκληρώνει τις εντολές πιο γρήγορα και μπορεί επίσης δυναμικά να κλείσει προσωρινά τις μονάδες που δε χρησιμοποιεί (π.χ clock gating). Η υλοποίηση τους επίσης εμπεριέχει σημαντικές προκλήσεις που αναγκάζουν τον σχεδιαστή να σκεφτεί τα πάντα πολύ διεξοδικά και έτσι να βελτιωθεί ο ίδιος και το επίπεδο γνώσεων του. Το μειονέκτημα τους βέβαια, όπως είναι φυσικό, είναι οτι ανεβάζουν επίσης κατακόρυφα το επίπεδο δυσκολίας σε σημείο μη εφικτό για ένα μάθημα εξαμήνου.

4.3 Συμπεράσματα για τις διάφορες τεχνικές BIST/SBIST που υλοποιήθηκαν

Συμπερασματικά για τις τρείς τεχνικές αυτοδοκιμής που υλοποιήθηκαν μπορούμε να πούμε πως μας επέτρεψαν να εξοικοιωθούμε με αυτές τις τεχνικές που δε τις είχαμε ξανασυναντήσει. Αν και η φύση των λαθών που ανιχνεύουν δε μας επέτρεψε να τις δούμε πραγματικά σε δράση, συνειδητοποιήσαμε πόσο σημαντικό είναι για ένα πολύπλοκο σχέδιο όπως είναι μια επεξεργαστική

μονάδα το να μπορεί να ανιχνεύει λάθη στο υλικό και να ειδοποιεί άμεσα το χρήστη για αυτά ώστε να επέμβει άμεσα πριν να είναι πολύ αργά.

Τέτοιες μέθοδοι κάνουν πιο αξιόπιστα τα συστήματα που αναπτύσουμε και είναι μονόδρομος για εφαρμογές υψηλού κινδύνου ή / και ρίσκου όπως είναι τα συστήματα αεροπλάνων, δορυφόρων, νοσοκομείων και γενικότερα οποιοδήποτε ηλεκτρονικό σύστημα λειτουργεί στους κλάδους των μεταφορών, του στρατού και της υγείας.