ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΗΜΜΥ

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΕΠΕΞΕΡΓΑΣΤΩΝ & ΥΛΙΚΟΥ

ΕΡΓΑΣΤΗΡΙΑΚΕΣ ΑΣΚΗΣΕΙΣ ΓΙΑ ΤΟ ΜΑΘΗΜΑ:
ΗΡΥ 302 – Οργάνωση Υπολογιστών

**ΕΑΡΙΝΟ ΕΞΑΜΗΝΟ 2025**

# Εργαστηριακή Άσκηση 1: Σχεδίαση Επεξεργαστή Ενός Κύκλου με τη Χρήση της $VHDL$

Ομάδα 24
Κύρκος Κωνσταντίνος: 2022030112
Μυλωνάκης Χαράλαμπος: 2022030133

## 1 Σύνοψη της Άσκησης

Ο σκοπός της πρώτης άσκησης ήταν η δημιουργία ενός επεξεργαστή ενός κύκλου με τη βοήθεια της $VHDL$. Ο επεξεργαστής αυτός μοιάζει αρκετά με τον $MIPS$, ωστόσο έχει διαφορετικό και πιο περιορισμένο $instruction\ fetch$. Η σχεδίασή του χωρίστηκε σε τρεις φάσεις για την καλύτερη κατανόηση της λειτουργίας του.

- Στην πρώτη φάση υλοποιήθηκε μια απλή μονάδα λογικών και αριθμητικών πράξεων ($ALU$), καθώς και το αρχείο των 32 καταχωρητών ($Register\ File$).
- Στη δεύτερη φάση σχεδίασης, υλοποιήσαμε τα βασικά μέρη του $Datapath$ του επεξεργαστή. Πιο συγκεκριμένα, ασχοληθήκαμε με την υλοποίηση των $DECSTAGE$, $IFSTAGE$, $EXECSTAGE$, και $MEMSTAGE$. Αφού προσομοιώσαμε το κάθε μέρος ξεχωριστά για να επιβεβαιώσουμε τη σωστή λειτουργία του, προχωρήσαμε στην τρίτη φάση της εργασίας.
- Στο τελευταίο στάδιο, χρησιμοποιήσαμε τα $components$ που σχεδιάσαμε στη δεύτερη φάση του $project$ και τα συνδέσαμε κατάλληλα σε ένα ενιαίο $datapath$. Έπειτα, δημιουργήσαμε το $ControlPath$, μια $FSM$, η ο-

ποία βάσει της εντολής που δέχεται καθορίζει τα σήματα ελέγχου για τα *components* του *datapath*.

- Τέλος, φορτώθηκαν τα *.coe* αρχεία που μας δόθηκαν, αλλά και όσα δημιουργήσαμε, ώστε να επιβεβαιώσουμε όλες τις λειτουργίες του επεξεργαστή σε ένα ενιαίο τελικό *testbench*.

# 2 Αναλυτική Περιγραφή

## 2.1 Πρώτη Φάση: *ALU* και *RegisterFile*

Η *ALU* παίρνει σαν είσοδο δύο 32 *bit* αριθμούς και ανάλογα ενός 4 *bit control signal* κάνει την ανάλογη πράξη. Ανάλογα τους δυο εισακτέους και την πράξη, πετάει 4 σήματα. Το αποτέλεσμα (*Output*), σήμα υπερχείλισης ($Overflow$), σήμα μηδενικού αποτελέσματος ($Zero - Out$), καθώς και κρατούμενο εξόδου ($Carry - Out$), εφόσον υπάρχει. Υπερχείλιση έχουμε μόνο στην πρόσθεση και στην αφαίρεση, όταν οι προσθετέοι είναι ομόσημοι μεταξύ τους αλλά το αποτέλεσμα είναι εταιρόσημό τους, ενώ στην αφαίρεση όταν η διαφορά μεταξύ δύο ετερόσημων αριθμών δίνει αριθμό με το πρόσημο του αφαιρετέου. Τέλος το $flag$ του $Carry-Out$ ανάβει στην περίπτωση που το 33ο *bit* του αποτελέσματος είναι διάφορο του μηδενός. Για την υλοποίηση της *ALU* χρησιμοποιήθηκαν οι βιβλιοθήκες $IEEE.NUMERIC_STD.ALL$ και $IEEE.STD_LOGIC_SIGNED.ALL$ για την υλοποίηση των πράξεων. Για την υλοποίηση του *register file* δημιουργήθηκαν 32 καταχωρητές των 32 *bit* χρησιμοποιώντας $for - generate$, με $WriteEnable$ και *Reset*, ένας *Decoder* 5 *bit* εισόδου και 32 *bit* εξόδου που «επιλέγει» σε ποιόν καταχωρητή να γράψει, γράφοντας σε αυτόν εφόσον υπάρχει ανοιχτό $WriteEnable$. Παράλληλα, δημιουργήθηκαν και δύο 32 *bit* πολυπλέκτες, που βάσει ενός *controlsignal* 5 *bit* ($address - read$) έβγαζαν την τιμή του κάθε καταχωρητή στην έξοδο. Στον πρώτο καταχωρητή ( *Register* 0 ) δόθηκε πάντα η τιμή 0. Μέσω ενδελεχούς *testbench* επιβεβαιώθηκε η σωστή λειτουργία τόσο της *ALU* όσο και του $REGISTER\ FILE$.

## 2.2 Δεύτερη Φάση: Υλοποίηση του *Datapath*

Στην δεύτερη φάση δημιουργήθηκαν τα 4 κύρια μέρη του *datapath*. Αρχικά υλοποιήθηκε το $EXECStage$. Η *ALU* ενώθηκε με έναν πολυπλέκτη δύο εισόδων 32 *bit* και εξόδου 32 *bit* με ένα *control signal* ($ALUBINSEL$), το οποίο καθορίζει αν η δεύτερη είσοδος της *ALU* θα πάρει είσοδο την δεύτερη έξοδο του $RegisterFile$ ή το επεξεργασμένο *immediate* κάποιας εντολής.
Έπειτα δημιουργήσαμε το $DECSTAGE$. Σε αυτό το στάδιο το $Instruction$ της *ROM* αποκωδικοποιείται καταλλήλως ώστε να γραφτεί στο αρχείο καταχωρητών. Πάντα ο πρώτος καταχωρητής ανάγνωσης είναι ο *RS* ενώ ο δεύτερος είναι είτε ο *RT* είτε ο *RD*, ανάλογα το $RF_BSEL$ (*control signal* πολυπλέκτη 2 σε 1) . Στην ουσία αυτός ο πολυπλέκτης διαχωρίζει τους δύο τύπος εντολών. Επίσης γίνεται η επεξεργασία του σήματος $IMMED$ 16 *bit* σε 32 *bit* ανάλογα την λειτουργία που χρειάζεται ($Zero-fill, Sign-extend, Zero-fill\ and shift, Sign-extend\ and$

$shift$). Τέλος τα δεδομένα που θα εγγραφούν στο $RegisterFile$ καθορίζονται από έναν 32 $bit$ πολυπλέκτη 2 σε 1, ο οποίος με το $control\ signal$ $RF_W R_D ATA_S EL$ επιλέγει αν θα γράψει από την $ALU$ ή από την $MEMORY$.
Εν συνεχεία, δημιουργήθηκε μέσω του $IpCoreGenerator$ της $XILINX$ μνήμη 1024 θέσεων βάθους 32 $bit$, για χρήση ως $RAM$ του επεξεργαστή. Η μνήμη είναι $ReadFirst$ και έχει μια θύρα για εγγραφή και μια για ανάγνωση, καθώς και $Write - Enable$.
Παράλληλα δημιουργήθηκε ένα ακόμη $Stage$ για την υλοποίηση της εντολής $loadword$ και $loadbyte$, το οποίο παίρνει ένα $control_signal$ για το αν θα βγάλει $byte\ or\ word$, ενώ αν θέλουμε $byte$ ο επεξεργαστής επιλέγει ποιο από τα 4 μέσω ενός ακόμη $control\ signal$, που παίρνει τα πρώτα δυο $bit$ του αποτελέσματος της ΑΛΥ.
Τέλος δημιουργήσαμε το $IF\ STAGE$. Εδώ διαβάζουμε μια εντολή ($Instruction$) από την $IMEM$ που δημιουργήσαμε στο $IpCoreGenerator$, 1024 θέσεων, βάθους 32 $bit$ τύπου $ROM$, και την φορτώνουμε στον $Program\ Counter$, έναν καταχωρητή ο οποίος «κρατάει την εντολή που πρόκειται να εκτελεστεί σε κάθε κύκλο του ρολογιού. Στον επόμενο κύκλο προχωράει στην επόμενη εντολή ($PC$+4). Στην περίπτωση που το $Instruction$ είναι $Branch$, $Branch\ Equal$, $Branch\ Not\ Equal$ ο $PC$ παίρνει την τιμή $PC$+4+$Immediate$ και το πρόγραμμα πάει σε αυτή την διεύθυνση

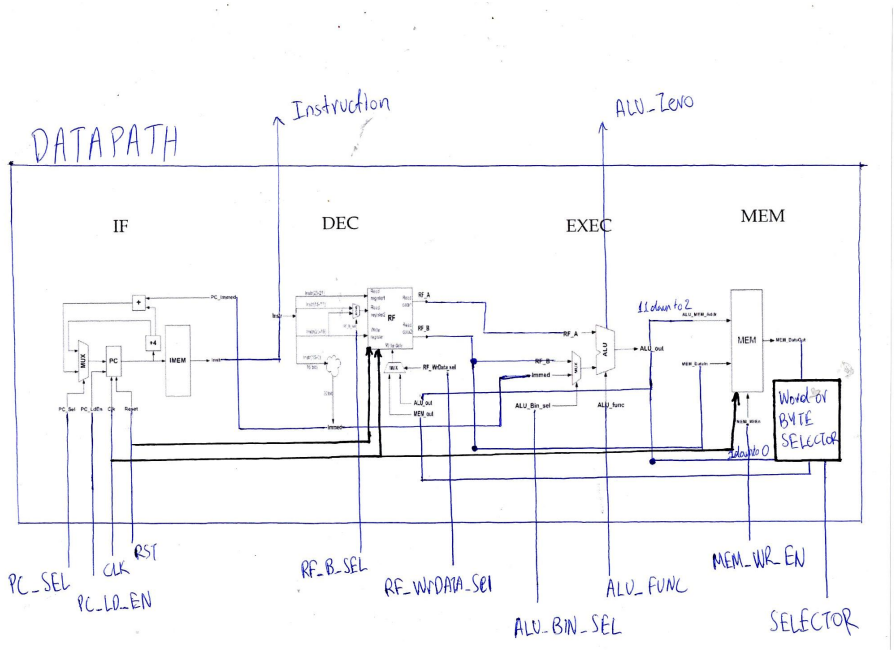
### 2.3 Τρίτη Φάση: Ολοκλήρωση του Επεξεργαστή

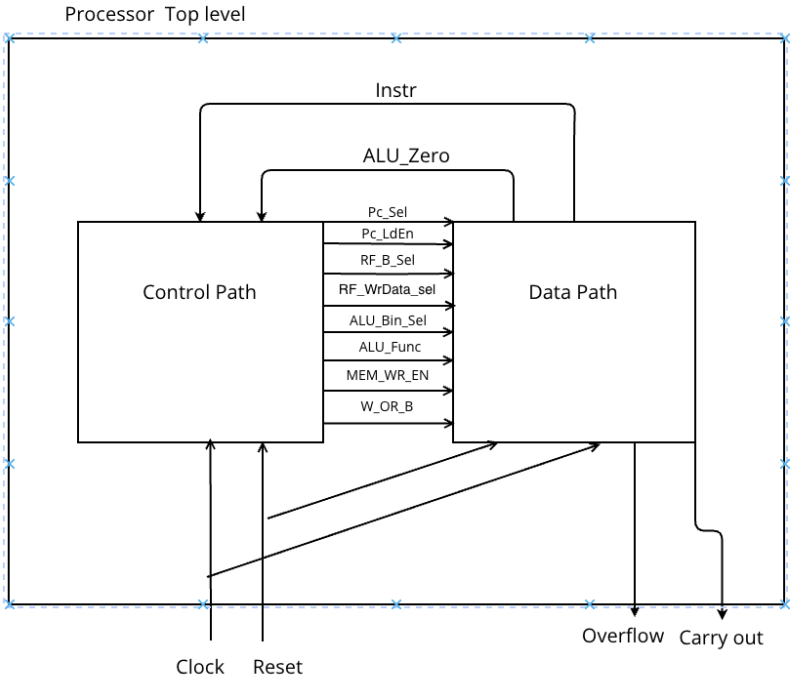
Στην Τρίτη φάση ενώθηκαν όλα τα προαναφερθέντα $Modules$ για την υλοποίηση ενός ενιαίου $datapath$, το οποίο παίρνει τις κατάλληλες εντολές από την $FSM$ στο $Control\ Path$ και είναι υπεύθυνο για την ορθή λειτουργία του επεξεργαστή μας. Παρακάτω φαίνεται το ενιαίο $DATAPATH$. Έπειτα, σχεδιάστηκε το $CONTROL\ path$, που όπως είπαμε είναι ουσιαστικά μια μηχανή πεπερασμένων καταστάσεων, υπεύθυνη για την παραγωγή σωστών σημάτων για είσοδο στο ΔΑΤΑΠΑΤΗ, παίρνοντας ως είσοδο το $Instruction$ της $ROM$ και το $Zero\ out$ από την $ALU$ για την υλοποίηση των εντολών $branch$. Το $Datapath$ και το $Control$ ενώθηκαν και δημιούργησαν το τελικό $TOP\ LEVEL\ file$, το οποίο έχει μόνο τις εισόδους του Ρολογιού και του $Reset$, καθώς όλη η επεξεργασία γίνεται από όλες τις κατώτερες βαθμίδες που προαναφέρθηκαν.

## 3 Πειράματα και Αποτελέσματα

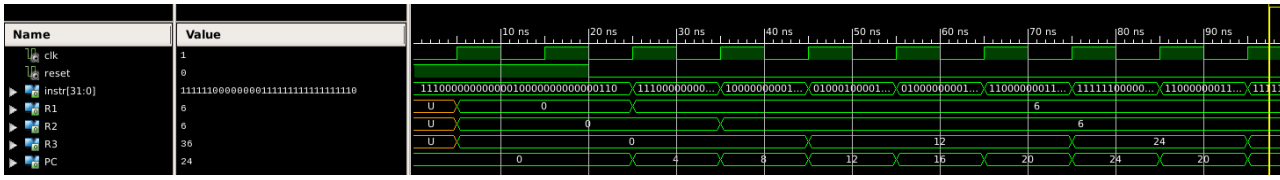
### 3.1 Πείραμα 1

```
li r1,6
li r2,6
add r1,r3,r2
bne r1,r2, 1
beq r1,r2, 0
addi r3,r3,12
```


DATAPATH
Instruction
ALU_Zero
IF
DEC
EXEC
MEM
PC_SEL
PC_LD_EN
CLK
RST
RF_B_SEL
RF_WrDATA_SEL
ALU_BIN_SEL
ALU_FUNC
MEM_WR_EN
SELECTOR
Word or BYTE SELECTOR


Processor Top level
Instr
ALU_Zero
Pc_Sel
Pc_LdEn
RF_B_Sel
RF_WrData_sel
ALU_Bin_Sel
ALU_Func
MEM_WR_EN
W_OR_B
Control Path
Data Path
Clock
Reset
Overflow
Carry out

Φορτώνει το 6 στον καταχωρητή 1
Φορτώνει το 6 στον καταχωρητή 2
Πρόσθεση καταχωρητών 1 και 2 και αποθήκευση στον 3
*equal* άρα $pc$+4
*equal* άρα $pc$+4+0
πρόσθεση του 12 στον $r3$ και καταχώρηση στον $r3$
($Pc$-4) *loop*

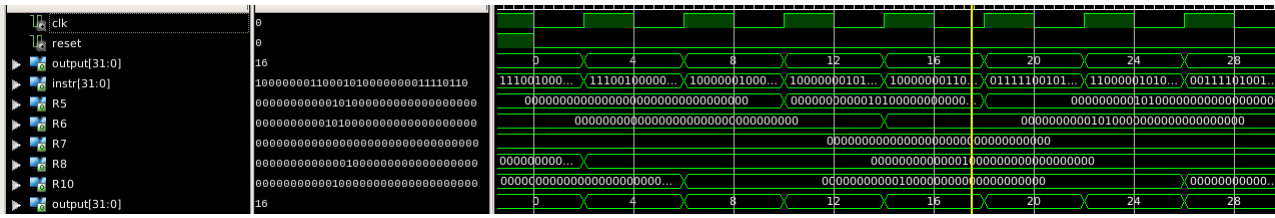



Επιβεβαιώθηκε η σωστή λειτουργία των εντολών φόρτωσης, πρόσθεσης και διακλάδωσης.

## 3.2 Πείραμα 2

```
lui r8,4
lui r10,16
or r8,r5,r10
rol r5,r6,1
sw r5,4(r10)
addi r10,4
lw r7,0(r10)
b -8
```

Φορτώνει *upper* 4 στον $R8$
Φορτώνει *upper* 16 στον $R10$
Φορτώνει στον $R5$ το αποτέλεσμα της $ALU(R8|R10)$
Φορτώνει στον $R6$ το αποτέλεσμα της $ALU(R5rol)$
Φορτώνει στον $R5$ το αποτέλεσμα της $ALU(R6rol)$περιμένουμε να παραμείνει ίδιο
Αποθηκεύει στη θέση μνήμης $[R5 + 4]$ την τιμή $R10$
Προσθέτουμε το $immed(4)$ στον $R10$
Φορτώνει στον $R7$ την τιμή της θέσης μνήμης $[R10 + 0]$
Μειώνει τον $PC$ πηγαίνοντας δυο εντολές πίσω Επαληθεύτηκε η σωστή λειτουργία των εντολών $OR$, $shift$, $store$ και $load$.

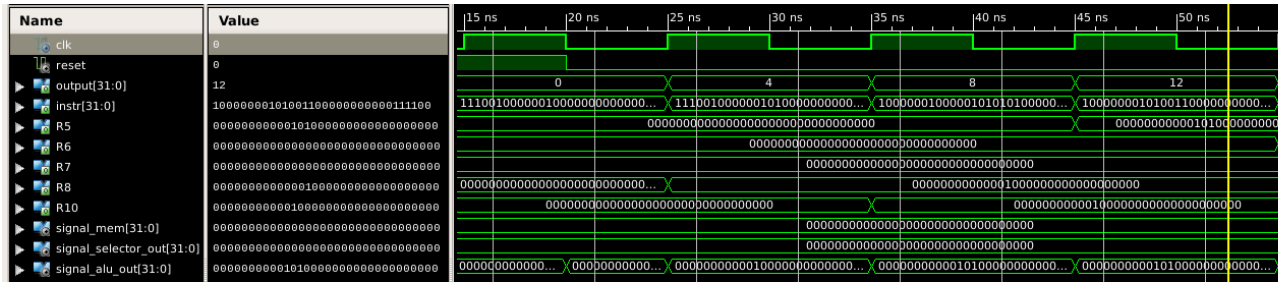


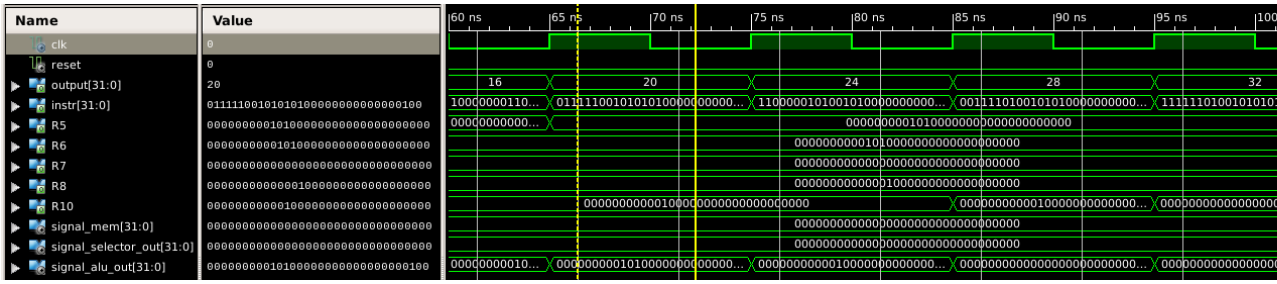

### 3.3 Πείραμα 3

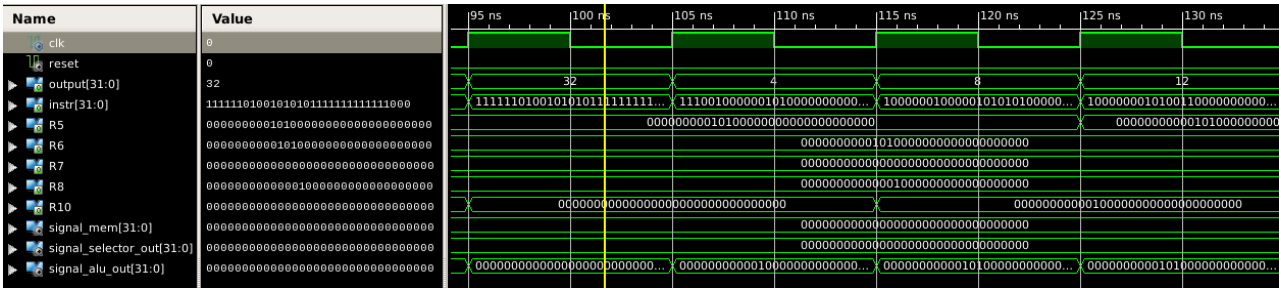
```
LI     R1,  6
LI     R2,  6
ADD  R1, R3, R2
SUB  R1, R3, R2
AND  R1, R4, R3
NOT  R1, R4
OR   R4, R5, R1
SRA  R5, R5
SLL  R5, R6
SRL  R5, R6
ROL  R5, R6
ROR  R6, R7
LUI   R10, 2
ADDI  R10, 4
ANDI R10, R12, 7
ORI R10, R12, 4
BEQ R10, R12, 2
BNE R10, R12, 1
LUI R0, R10, 6
SW R5   R10  0
LB R10, R14, 0
B -2
```

Φορτώνει το 6 στον καταχωρητή 1
Φορτώνει το 6 στον καταχωρητή 2
Φορτώνει $R1 + R2$ στον $R3$
Φορτώνει $R1 - R2$ στον $R3$
Φορτώνει $R1R3$ στον $R4$
Φορτώνει $!R1$ στον $R4$
Φορτώνει $R4|R1$ στον $R5$
$SHIFT\ R5$
$SHIFT\ LEFT\ R5$ ΚΑΙ ΤΟ ΒΑΖΕΙ ΣΤΟΝ $R6$
$SHIFT\ LEFT\ R5$ ΚΑΙ ΤΟ ΒΑΖΕΙ ΣΤΟΝ $R6$
$ROL\ LEFT\ R5$ ΚΑΙ ΤΟ ΒΑΖΕΙ ΣΤΟΝ $R6$
$ROTATE\ RIGHT\ R6$ ΚΑΙ ΤΟ ΒΑΖΕΙ ΣΤΟΝ $R7$
$LOAD\ IMMED$ και το φορτώνει στον $R10$

$ADD\ Immed(+4)$ στον $R10$
$Immed(7)\ AND\ R10$ και το φορτώνει στον $R12$
$Immed(4)\ ORI\ R10$ και το φορτώνει στον $R12$
Αν ισχύει η ισότητα $PC + 4 + SignExtend(Imm)$ 2 αλλιώς προχωράει κανονικά.(Εδω,περιμένουμε να μην ισχύει)
Αν ισχύει η ανισότητα $PC{+}4{+}SignExtend(Imm)$ 2 αλλιώς προχωράει κανονικά.(Εδω,περιμένουμε να ισχύει)
Φορτώνει το $UPPER6$ στον καταχωρητή $R10$
Αποθηκέυει στην θέση μνήμης $[R5 + 0]$ την τιμή του P10
Φορτώνει το $4oBYTE$ από την θέση μνήμης $[timR10{+}immed]$ στον καταχωρητή $R15$ Μείωνουμε τον $PC$ και ξανα εκτελούμε τις τελευταίες εντολές




Πραγματοποιήθηκε εκτενής έλεγχος όλων των υποστηριζόμενων εντολών, συμπεριλαμβανομένων λογικών πράξεων, διακλαδώσεων και άμεσης φόρτωσης.

# 4 Σύνοψη

Η εργαστηριακή άσκηση μάς βοήθησε να κατανοήσουμε εις βάθος την αρχιτεκτονική ενός επεξεργαστή ενός κύκλου. Επιπλέον, μάθαμε να εντοπίζουμε και να διορθώνουμε σφάλματα, καθώς και να διαχειριζόμαστε τον σχεδιασμό πολύπλοκων κυκλωμάτων σε $VHDL$.