ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ,
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ, ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ ΚΑΙ ΔΙΚΤΥΩΝ

Σύνθεση & Ανάλυση Dual-Core MIPS 32-bit επεξεργαστή

Αικατερίνη Γρατσία

Διπλωματική εργασία

Επιβλέποντες καθηγητές:
Γεώργιος Σταμούλης – Καθηγητής

Νέστορας Ευμορφόπουλος – Λέκτορας

Βόλος, Φεβρουάριος 2011

Σύνθεση & Ανάλυση Dual-Core MIPS 32-bit επεξεργαστή

Γεώργιος Σταμούλης – Καθηγητής

Νέστορας Ευμορφόπουλος – Λέκτορας

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή μου και πρόεδρο της Σχολής, κ. Σταμούλη Γεώργιο για τη βοήθεια του στην ολοκλήρωση της διπλωματικής μου εργασίας αλλά κυρίως για τη βοήθεια και τη στήριξη που μου πρόσφερε σε όλα τα χρόνια των σπουδών μου. Επίσης ευχαριστώ τον καθηγητή μου κ. Ευμορφόπουλο Νέστορα.

Ένα μεγάλο ευχαριστώ στον Γεώργιο Παναγόπουλο, που για εμένα αποτελεί πρότυπο ανθρώπου και επιστήμονα. Τον ευχαριστώ για την αγάπη του και που βρίσκεται πάντα δίπλα μου να με στηρίζει. Οι συμβουλές του είναι για εμένα οδηγός.

Ευχαριστώ από καρδιάς τους γονείς μου, Νικόλα και Μαρία, που πάντα πίστευαν σε εμένα. Τους ευχαριστώ για τη βοήθειά τους σε όλους τους τομείς, αλλά και για την ελευθερία που είχα πάντα στη ζωή μου να κάνω μόνη μου τις επιλογές μου έχοντας εκείνους υποστηρικτές.

Ευχαριστώ την αδερφή μου Κυριακή Γρατσία, που είναι από τους σημαντικότερους και πολυτιμότερους ανθρώπους στη ζωή μου και την αγαπημένη μου φίλη και συμφοιτήτρια Δήμητρα Κατάνου, για τις στιγμές που ζήσαμε όλες μαζί στα φοιτητικά μας χρόνια στο Βόλο καθώς και για την τόσο δυνατή και αληθινή φιλία που έχουμε.

Στον Γιώργο μου

Πίνακας Περιεχομένων

Πίνακας Περιεχομένων 5

1. Εισαγωγή 7
2. Βασικό σύνολο εντολών 10
   - 2.1 Καταχωρητές 10
3. Testbench 12
4. Single-Cycle processor 15
   - 4.1 Βασικές Δομικές Μονάδες 15
     - 4.1.1 Φάκελος Καταχωρητών (Register File) 15
     - 4.1.2 Αριθμητική Λογική Μονάδα (ALU) 16
     - 4.1.3 Μονάδα ελέγχου (Controller) 18
     - 4.1.4 Μετρητής Εντολών (Program Counter) 19
     - 4.1.5 Μνήμη Εντολών & Δεδομένων & (Instruction & Data Cache) 20
   - 4.2 Προσομοίωση 21
     - 4.2.1 Προσομοίωση σε Λογικό επίπεδο 21
     - 4.2.2 Σύνθεση και Χρονική Ανάλυση 22
   - 4.3 Συμπεράσματα 25
5. Pipeline processor 26
   - 5.1 Βασικές Δομικές Μονάδες 26
     - 5.1.1 Μνήμη Εντολών (Instruction Cache) 26
     - 5.1.2 Μνήμη Δεδομένων (Data Cache) 28
     - 5.1.3 Διαχειριστής Μνήμης (Arbitrator) 30
   - 5.2 Προσομοίωση 31
     - 5.2.1 Προσομοίωση σε λογικό επίπεδο 31
     - 5.2.2 Σύνθεση και Χρονική Ανάλυση 33
   - 5.3 Συμπεράσματα 35
6. Multi-core processor 36
   - 6.1 Coherence Controller 37
   - 6.2 Μνήμη δεδομένων για dual-core επεξεργαστή 38
   - 6.3 Σχηματικό Διάγραμμα του Dual-Core 38
   - 6.4 Το παράλληλο πρόγραμμα mergesort 39

6.5 Προσομοίωση ... 45
6.6 Συμπεράσματα ... 47
7. FPGA ... 48
8. Αποτελέσματα & Απόδοση των CPU ... 53
A. Βασικές κυκλωματικές δομές σε VHDL ... 54
A.1 Πολυπλέκτης (MUX) ... 54
A.2 D Flip-Flop ... 55
B. Αναφορές ... 56

1. Εισαγωγή

Κάθε επιτραπέζιο υπολογιστικό σύστημα, laptop, server, κινητό τηλέφωνο όπως και πολλές άλλες ηλεκτρονικές συσκευές έχουν ως κεντρική μονάδα επεξεργασίας των δεδομένων τους επεξεργαστές γενικού σκοπού (general purpose CPUs). Ο επεξεργαστής είναι ένα ψηφιακό κύκλωμα το οποίο διαβάζει και εκτελεί εντολές. Οι εντολές αυτές «λένε» στον επεξεργαστή τι να κάνει. Αυτές οι εντολές είναι συνήθως πάρα πολύ απλές όπως διάβασε δεδομένα από τη μνήμη, γράψε δεδομένα στη μνήμη ή πρόσθεσε δύο αριθμούς. Όμως αυτές οι απλές πράξεις πρέπει να εκτελούνται πάρα πολύ γρήγορα έτσι ώστε ο χρήστης του επεξεργαστή ή του συστήματος που χρησιμοποιεί τον επεξεργαστή να βλέπει μια ομαλή εκτέλεση του προγράμματος που εκτελείται.

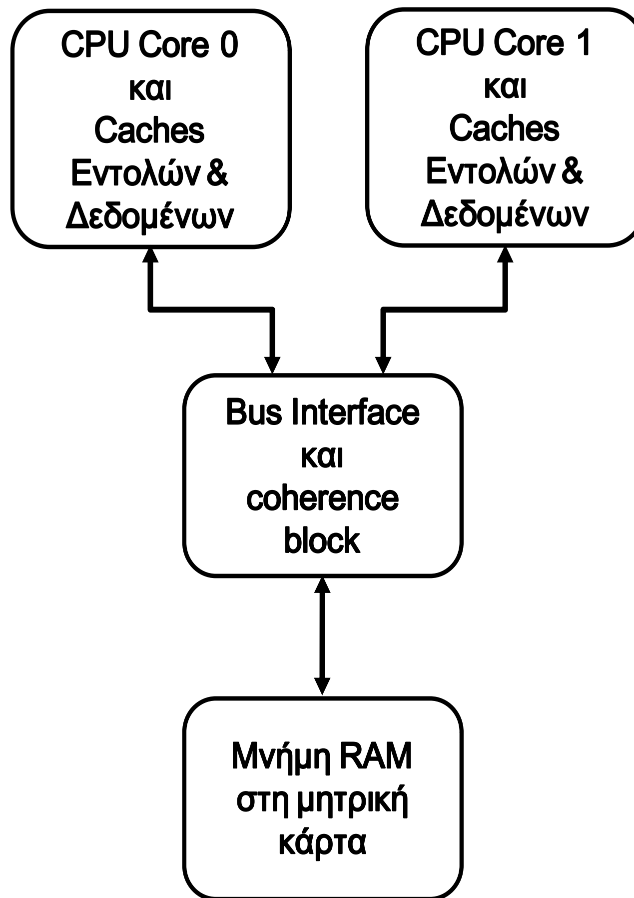
Ένας τρόπος για να επιτύχουμε βελτίωση του χρόνου εκτέλεσης είναι να χρησιμοποιήσουμε μικρότερα τρανζίστορ, δηλαδή να πάμε σε καινούργιες τεχνολογίες. Όπως φαίνεται στο Πίν. 12.1-1 η εταιρία μικροεπεξεργαστών Intel πηγαίνοντας σε μικρότερες τεχνολογίες όχι μόνο κατάφερε να επιταχύνει τους επεξεργαστές της (νόμος του Moore: διπλασιασμός της ταχύτητας κάθε 1.5 χρόνο) αλλά και να χωρέσει περισσότερα τρανζίστορ στο ίδιο die. Περισσότερα τρανζίστορ σημαίνει ότι περισσότερες λειτουργίες μπορεί να εκτελέσει ο επεξεργαστής.

Ένας άλλος τρόπος για να αυξήσουμε την ταχύτητα εκτέλεσης εντολών είναι να χρησιμοποιήσουμε περισσότερους του ενός core σε κάθε die (Core είναι το τμήμα του επεξεργαστή που διαβάζει δεδομένα από τη μνήμη, τα επεξεργάζεται και τα ξαναγράφει στη μνήμη). Οι επεξεργαστές αρχικά είχαν μόνο έναν core. Ένας dual-core επεξεργαστής έχει δύο cores οι οποίοι επεξεργάζονται παράλληλα τα δεδομένα των προγραμμάτων με αποτέλεσμα την επιτάχυνση της εκτέλεσής τους. Οι σημερινοί επεξεργαστές έχουν ενσωματωμένους περισσότερους από 40 cores σε κάθε die (multi-cores). Είναι όμως σημαντικό να σημειωθεί ότι σε έναν επεξεργαστή multi-core το κάθε core δεν είναι ανεξάρτητο από τα άλλα αφού όλα τα core προσπελαύνουν μια κοινή μνήμη (RAM). Έτσι αν κάποιο core έχει μεταβάλλει τα δεδομένα στη μνήμη τότε το δεύτερο core θα πρέπει να είναι ενήμερο γι αυτήν τη πράξη και αν χρειαστεί να χρησιμοποιήσει αυτά τα δεδομένα θα πρέπει να διαβάσει ένα σωστό αντίγραφο. Ένα γενικό διάγραμμα της διαδικασίας που μόλις περιγράψαμε φαίνεται σχηματικά στο διάγραμμα της Σχ. 2.1-1. Αυτό είναι ένα κλασσικό πρόβλημα καταναλωτή – παραγωγού που όπως θα δούμε λύνεται με τη χρήση του πρωτοκόλλου MSI.

Name	Date	Transistors	Microns	Clock speed	Data width	MIPS
8080	1974	6000	6	2MHz	8 bits	0.64
8088	1979	29000	3	5 MHz	16 bits, 8-bit bus	0.33
80286	1982	134000	1.5	6 MHz	16 bits	1
80386	1985	275000	1.5	16 MHz	32 bits	5
80486	1989	1200000	1	25 MHz	32 bits	20
Pentium	1993	3100000	0.8	60 MHz	32 bits, 64-bit bus	100
Pentium II	1997	7500000	0.35	233 MHz	32 bits, 64-bit bus	~300
Pentium III	1999	9500000	0.25	450 MHz	32 bits, 64-bit bus	~510
Pentium 4	2000	42000000	0.18	1.5 GHz	32 bits, 64-bit bus	~1700
Pentium 4 "Prescott"	2004	125000000	0.09	3.6 GHz	32 bits, 64-bit bus	~7000

Πίν. 12.1-1: Η Intel κινήθηκε σε μικρότερες τεχνολογίες κατά τη διάρκεια των χρόνων.

Έτσι σε αυτή τη διπλωματική εργασία θα υλοποιήσουμε τρεις διαφορετικούς τύπους επεξεργαστών: (α) έναν **single-cycle**, (β) έναν **pipeline** και (γ) έναν **dual core**. Σκοπός μας είναι αρχικά η ορθή λειτουργικότητα όλων των επεξεργαστών και έπειτα η πραγματική τους υλοποίηση



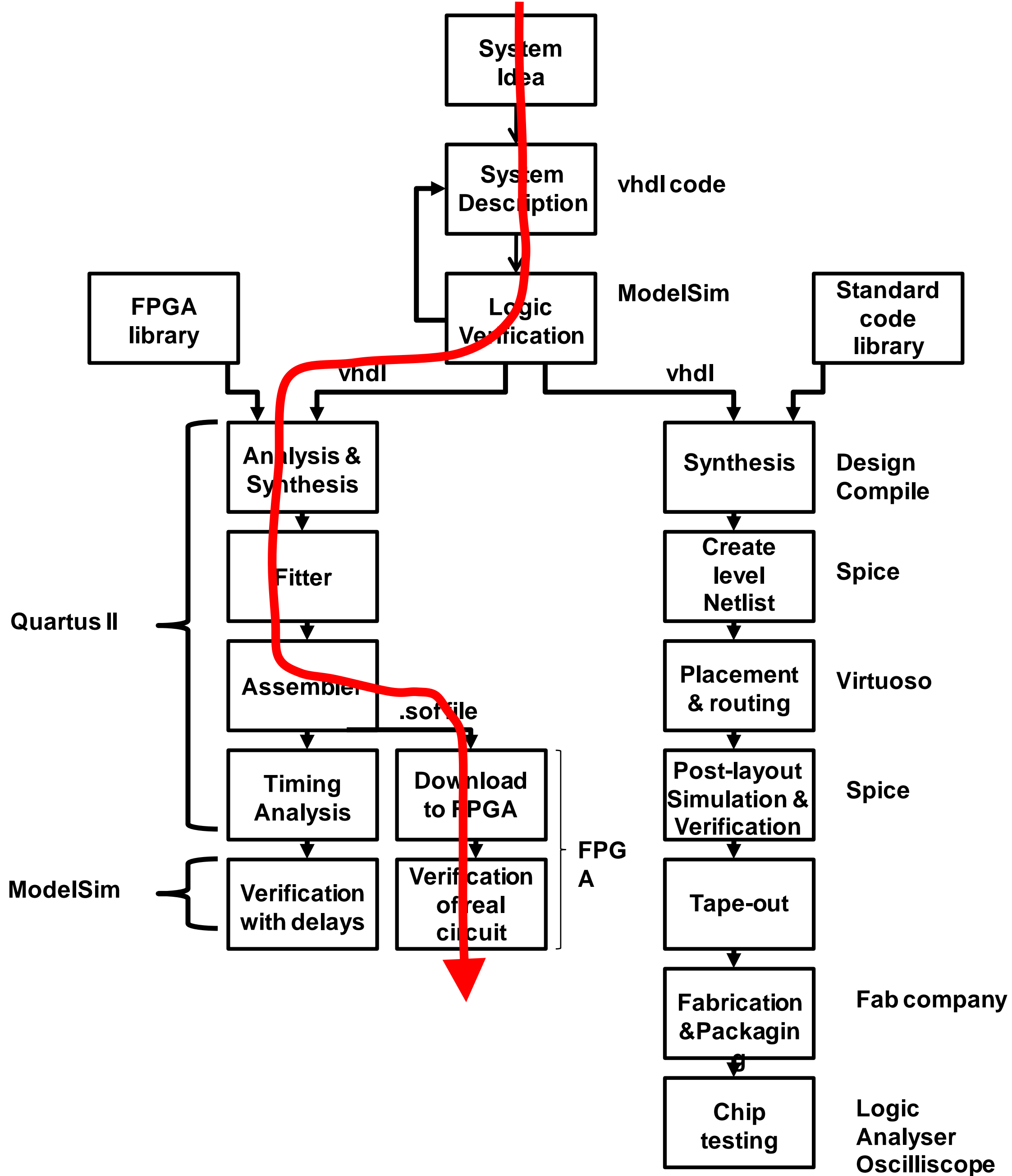
Σχ. 2.1-1:Γενικό σχηματικό διάγραμμα της επικοινωνίας των cores ενός επεξεργαστή dual-core. Το κοινό τμήμα των cores είναι η μνήμη RAM και η επικοινωνία γίνεται μέσω του bus.

σε FPGA. Έχοντας υλοποιήσει τους επεξεργαστές σε FPGA θα μπορέσουμε να μετρήσουμε την απόδοσή τους και να τους συγκρίνουμε μεταξύ τους.

Η περιγραφή όλων των κυκλωμάτων γίνεται με τη χρήση της γλώσσας VHDL. Η VHDL είναι μια γλώσσα περιγραφής ολοκληρωμένων ψηφιακών κυκλωμάτων που χρησιμοποιείται για να περιγράψει συστήματα τα οποία θα υλοποιηθούν είτε σε FPGA (Field Programmable Gate Array) είτε σε ολοκληρωμένο κύκλωμα IC (integrated circuit). Στο Σχ. 2.1-2 δείχνουμε τα διαγράμματα ροής για την υλοποίηση ψηφιακών κυκλωμάτων ξεκινώντας από την περιγραφή τους σε γλώσσα VHDL και καταλήγοντας στην υλοποίηση τους σε hardware. Σε κάθε βήμα της διαδικασίας υλοποίησης αναφέρουμε παραδείγματα εργαλείων και προγραμμάτων που μπορούν να χρησιμοποιηθούν για την παραγωγή του τελικού κυκλώματος. Σε αυτή την εργασία το τελικό κύκλωμα υλοποιείται σε FPGA, δηλαδή ακολουθήθηκε το διάγραμμα ροής που φαίνεται στην εικόνα με κόκκινο βέλος.

Σύμφωνα με αυτό το διάγραμμα ο επεξεργαστής περιγράφεται σε VHDL και επαληθεύεται σε λογικό επίπεδο με το πρόγραμμα ModelSim. Όταν είμαστε σίγουροι για τη λογική ορθότητα του κώδικα τότε είμαστε έτοιμοι για την ανάλυση και σύνθεση του κυκλώματος. Το βήμα αυτό προϋποθέτει τη χρήση μιας βιβλιοθήκης που δίδεται από το FPGA και περιλαμβάνει πληροφορίες για τις καθυστερήσεις των πυλών του FPGA. Αυτό το στάδιο όπως και τα υπόλοιπα εκτελούνται από το πρόγραμμα Quartus II. Ο Fitter είναι υπεύθυνος για το placement και routing του FPGA και ο Assembler

για να συλλέξει όλη τη χρήσιμη πληροφορία και να δημιουργήσει το αρχείο .sof. Το αρχείο .sof είναι αυτό που χρησιμοποιούμε για να κατεβάσουμε το κύκλωμά μας στο FPGA όπου πλέον ακολουθεί η επαλήθευση της ορθής του λειτουργίας. Επιπλέον η ορθή λειτουργία του κυκλώματος μπορεί να επαληθευτεί με το ModelSim με τη χρήση της VHDL που παράχθηκε μετά τη χρονική ανάλυση. Αυτή η VHDL περιγράφει το κύκλωμα που δόθηκε αρχικά αλλά περιλαμβάνει επιπλέον πληροφορία για τις καθυστερήσεις των πυλών που είναι κατάλληλη για την επαλήθευση της ορθής χρονικής ανάλυσης.



Σχ. 2.1-2: Διάγραμμα ροής υλοποίησης ψηφιακών κυκλωμάτων.

2. Βασικό σύνολο εντολών

Σε αυτήν την ενότητα θα περιγράψουμε το βασικό σύνολο εντολών που αντιλαμβάνονται οι επεξεργαστές MIPS οι οποίοι υλοποιήθηκαν σε αυτήν την εργασία.

2.1 Καταχωρητές

Ο φάκελος καταχωρητών αποτελείται από 32 καταχωρητές που συμβολίζονται ως \$0 - \$31 των 32bit ο καθένας. Ο καταχωρητής 0 έχει μόνιμα την τιμή 0 και ο καταχωρητής 1 είναι δεσμευμένος για τον assembler (συμβολομεταφραστής). Οι καταχωρητές \$4 - \$7(\$a0 - \$a3) χρησιμοποιούνται ως ορίσματα συναρτήσεων και οι καταχωρητές \$8 - \$15(\$t0 - \$t7) ως temporary καταχωρητές. Ο \$29(\$sp) είναι ο stack pointer,δηλαδή δείχνει στην αρχή της στοίβας εκτέλεσης. Τέλος ο \$31(\$ra) περιλαμβάνει τη διεύθυνση επιστροφής της συνάρτησης που τον κάλεσε.

Όλες οι εντολές μηχανής του επεξεργαστή MIPS έχουν σταθερό μήκος 32bit ο καθένας. Οι εντολές ταξινομούνται σε 3 μεγάλες κατηγορίες R-type, I-type και J-type. Στη συγκεκριμένη υλοποίηση οι R-type εντολές που υλοποιούνται δίνονται στους 2.1-1:.

Εντολή Μηχανής	Assembly	Ψευδοκώδικας
ADDU	\$rd,\$rs,\$rt	R[rd] <= R[rs] + R[rt]
AND	\$rd,\$rs,\$rt	R[rd] <= R[rs] AND R[rt]
JR	\$rs	PC <= R[rs]
NOR	\$rd,\$rs,\$rt	R[rd] <= ~(R[rs] OR R[rt])
OR	\$rd,\$rs,\$rt	R[rd] <= R[rs] OR R[rt]
SLT	\$rd,\$rs,\$rt	R[rd] <= (R[rs] < R[rt]) ? 1 : 0
SLTU	\$rd,\$rs,\$rt	R[rd] <= (R[rs] < R[rt]) ? 1 : 0
SLL	\$rd,\$rs,shamt	R[rd] <= R[rs] << shamt
SRL	\$rd,\$rs,shamt	R[rd] <= R[rs] >> shamt
SUBU	\$rd,\$rs,\$rt	R[rd] <= R[rs] - R[rt]
XOR	\$rd,\$rs,\$rt	R[rd] <= R[rs] XOR R[rt]

2.1-1: R-type εντολές που υλοποιήθηκαν στον επεξεργαστή MIPS.

ADDIU	\$rt,\$rs,imm	R[rt] <= R[rs] +SignExtImm
ANDI	\$rt,\$rs,imm	R[rt] <= R[rs] + ZeroExtImm
BEQ	\$rs,\$rt,label	PC <= (R[rs] == R[rt]?npc+BranchAddr:npc
BNE	\$rs,\$rt,label	PC <= (R[rs] != R[rt])?npc+BranchAddr:npc
LUI	\$rt,imm	R[rt] <= {imm,16b'0}
LW	\$rt,imm(\$rs)	R[rt] <= M[R[rs] + SignExtImm]
ORI	\$rt,\$rs,imm	R[rt] <= R[rs] OR ZeroExtImm
SLTI	\$rt,\$rs,imm	R[rt] <= (R[rs] < SignExtImm)?1:0
SLTIU	\$rt,\$rs,imm	R[rt] <= (R[rs] < SignExtImm)?1:0
SW	\$rt,imm(\$rs)	M[R[rs] + SignExtImm] <= R[rt]
TSL	\$rt,imm(\$rs)	R[rt] <= M[R[rs] + SignExtImm] <= 0xffffffff
XORI	\$rt,\$rs,imm	R[rt] <= R[rs] XOR ZeroExtImm

Πίν. 2.1-2: I-type εντολές που υλοποιήθηκαν στον επεξεργαστή MIPS.

J	Label	PC <= JumpAddr
JAL	Label	R[31] <= npc; PC <= JumpAddr

2.1-3: J-type εντολές που υλοποιήθηκαν στον επεξεργαστή MIPS.

Τέλος, για αυτή την εργασία ορίζουμε κάποιες ειδικές εντολές (halt, org, chw και cfw). Η εντολή halt μεταφράζεται στην εντολή FFFFFF η οποία προτρέπει τον επεξεργαστή να τερματίσει την λειτουργία του προγράμματος (δες Κεφ. 3 για λεπτομέρειες). Η εντολή org θέτει τη διεύθυνση στη μνήμη εντολών όπου θα τοποθετηθεί ο κώδικας που την ακολουθεί. Οι εντολές chw και cfw θέτουν στη μνήμη δεδομένων δεδομένα μισής και ολόκληρης λέξης. Σημειώστε ότι οι εντολές org, chw και cfw είναι υπεύθυνες για την αρχικοποίηση των μνημών (εντολών και δεδομένων) του επεξεργαστή.

3. Testbench

Η διαδικασία της απόδειξης της ορθότητας ενός μεγάλου και πολύπλοκου κυκλώματος όπως είναι οι επεξεργαστές που υλοποιήσαμε σε αυτή την εργασία είναι γενικά μια επίπονη διαδικασία. Για τον λόγο αυτό αποφασίσαμε να δημιουργήσουμε ένα testbench το οποίο να έχει τη δυνατότητα να ελέγχει εύκολα και κομψά τους επεξεργαστές για πολλές περιπτώσεις. Σε αυτό το κεφάλαιο θα περιγράψουμε αυτή τη μέθοδο.

Το testbench είναι ένα εικονικό περιβάλλον που έχει σκοπό να ελέγξει και να πιστοποιήσει την ορθή λειτουργία του μοντέλου του επεξεργαστή. Ειδικότερα, το testbench λαμβάνει ως είσοδο ένα αρχείο (meminit.hex) το οποίο περιλαμβάνει τις εντολές (instructions) του προγράμματος που θα εκτελεστεί από τον επεξεργαστή καθώς και τις αντίστοιχες διευθύνσεις τους. Έτσι οι εντολές αυτές τοποθετούνται στις κατάλληλες διευθύνσεις στη μνήμη εντολών (instruction cache) του επεξεργαστή. Στο τέλος αυτού του βήματος ο επεξεργαστής έχει αρχικοποιηθεί με το πρόγραμμα που βρίσκεται στο meminit.hex. Θα ήταν χρήσιμο να επισημανθεί ότι το αρχείο meminit.hex τελικά είναι είσοδος στη μνήμη RAM που μας έχει δοθεί από το FPGA και πρακτικά βρίσκεται στη μητρική κάρτα του υπολογιστικού συστήματος. Για παράδειγμα, παρακάτω παραθέτουμε το port map όπου έχουμε επισημάνει την είσοδο του αρχείου:

```
altsyncram_component : altsyncram
GENERIC MAP (
      clock_enable_input_a => "BYPASS",
      clock_enable_output_a => "BYPASS",
      init_file => "meminit.hex",
      intended_device_family => "Cyclone II",
      lpm_hint => "ENABLE_RUNTIME_MOD=YES, INSTANCE_NAME=ramM",
      lpm_type => "altsyncram",
      numwords_a => 8192,
      operation_mode => "SINGLE_PORT",
      outdata_aclr_a => "NONE",
      outdata_reg_a => "UNREGISTERED",
      power_up_uninitialized => "FALSE",
      widthad_a => 13,
      width_a => 32,
      width_byteena_a => 1
)
PORT MAP (
      wren_a => wren,
      clock0 => nclk,
      address_a => new_addr,
      data_a => data,
      q_a => sub_wire0
);
```

Η οργάνωση του αρχείου meminit.hex φαίνεται από παρακάτω παράδειγμα:

```
0000340F0080
0100ADE00008
020034010001
030034020002
04008DE50000
05008DE60004
060000003825
```

```
070000004025
080000A13824
090010E00001
0A0001064021
0B0000C03040
0C0000A02842
0D0010A00001
0E001000FFF9
0F00ADE80008
1000FFFFFFFF
200000000005
21000000000A
```

Τα δεδομένα στο αρχείο εισόδου αναπαρίστανται σε δεκαεξαδικό σύστημα και κάθε γραμμή του αρχείου αναφέρεται σε μια εντολή. Επίσης κάθε γραμμή του αρχείου μπορεί να διασπαστεί σε τρία πεδία. Το πρώτο πεδίο αποτελείται από τα πρώτα τέσσερα ψηφία τα οποία αναφέρονται στη διεύθυνση της εντολής, δηλαδή η διεύθυνση εισόδου της RAM θα έχει μήκος 16bit. Παρακάτω παραθέτουμε το entity της RAM όπου επισημαίνουμε ότι όντως η διεύθυνση εισόδου της RAM είναι 16bit:

```
ENTITY ram IS
   PORT
   (
        address   : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
        clock     : IN STD_LOGIC ;
        data      : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        wren      : IN STD_LOGIC ;
        q         : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
   );
END ram;
```

Το δεύτερο πεδίο αποτελείται από τα 8 επόμενα ψηφία της γραμμής και αναφέρεται στην εντολή. Έτσι όπως ήταν αναμενόμενο μια εντολή έχει μήκος 32bit (αφού υλοποιούμε 32-bit επεξεργαστή). Για παράδειγμα, η πρώτη γραμμή προτρέπει το testbench να τοποθετήσει την εντολή **340F0080** στη διεύθυνση x00 (b00000000). Με τον ίδιο τρόπο, η δεύτερη γραμμή θα τοποθετήσει την εντολή **ADE00008** στη διεύθυνση x01 (b00000001). Σημειώστε ότι οι διευθύνσεις αναφέρονται σε λέξεις (4bytes) και όχι σε bytes. Έτσι αν θέλουμε να βρούμε τις διευθύνσεις σε bytes πρέπει να πολλαπλασιάζουμε τις διευθύνσεις του αρχείου 4. Το τελευταίο πεδίο αποτελείται από τα τελευταία δύο ψηφία της γραμμής. Αυτό το πεδίο δεν προσφέρει κάποια επιπλέον πληροφορία για την εντολή αλλά χρησιμοποιείται από τη μνήμη RAM για ανάκτηση δεδομένων σε περίπτωση που κάποια bit καταστραφούν κατά την ανάγνωση του αρχείου. Αυτή η τεχνική συνήθως αναφέρεται ως cyclic redundancy check (CRC). Με αυτό το πεδίο δεν θα ασχοληθούμε άλλο σε αυτή τη διπλωματική. Τελευταία εντολή του προγράμματος είναι η **FFFFFFFF**. Όπως αναφέραμε στο Κεφ. 2 αυτή η εντολή προτρέπει τον επεξεργαστή να τερματίσει (halt). Τέλος, εκτός από εντολές το αρχείο εισόδου περιέχει και δεδομένα που πιθανώς θα χρειαστούν κατά την εκτέλεση του προγράμματος. Στη συγκεκριμένη περίπτωση οι τελευταίες δύο γραμμές (με κόκκινο) είναι τα δεδομένα μαζί με τις αντίστοιχες διευθύνσεις τους. Προφανώς, τα δεδομένα αυτά θα φορτωθούν από εντολές lw.

Η έξοδος του testbench είναι ένα άλλο αρχείο, το memout.hex, που έχει δομή ίδια με αυτή που περιγράψαμε για το meminit.hex και αναπαριστά τα δεδομένα της μνήμης RAM μετά τον τερματισμό της εκτέλεσης του προγράμματος. Άρα το αρχείο αυτό περιλαμβάνει όχι μόνο τις εντολές αλλά τα δεδομένα που αποθηκεύτηκαν κατά τη διάρκεια της εκτέλεσης του προγράμματος, δηλαδή από τις εντολές sw. Η έξοδος του αρχείου meminit.hex που παρουσιάστηκε παραπάνω είναι η ακόλουθη:

```
0000340F0080
0100ADE00008
020034010001
030034020002
04008DE50000
05008DE60004
060000003825
070000004025
080000A13824
090010E00001
0A0001064021
0B0000C03040
0C0000A02842
0D0010A00001
0E001000FFF9
0F00ADE80008
1000FFFFFFFF
200000000005
21000000000A
220000000032
```

Το πρόγραμμα αυτό εκτελεί την πράξη του πολλαπλασιασμού. Στη συγκεκριμένη περίπτωση πολλαπλασιάζει τον αριθμό x05 (5) με τον x0A (10) και το αποτέλεσμα είναι προφανώς x32 (50). Με αυτό τον τρόπο μπορούμε εύκολα να ελέγχουμε αν ο επεξεργαστής λειτουργεί σωστά χωρίς να χρειαστεί να ελέγχουμε βήμα-βήμα την εκτέλεση των εντολών που με τον απλό πολλαπλασιασμό που παρουσιάσαμε παραπάνω ή και για πιο μεγάλα προγράμματα είναι μια επίπονη διαδικασία.

4. Single-Cycle processor

Ένας single cycle επεξεργαστής ολοκληρώνει την εκτέλεση μίας εντολής σε έναν κύκλο μηχανής μέσω μίας διαδρομής δεδομένων που θα αναλύσουμε παρακάτω. Περιληπτικά αυτό περιλαμβάνει την προσκόμιση της εντολής από τη μνήμη εντολών (RAMI), την αποκωδικοποίηση της, την εκτέλεση της στην αριθμητική/λογική μονάδα (ALU), την προσπέλαση της μνήμης δεδομένων, αν αυτό είναι απαραίτητο και την επανεγγραφή των αποτελεσμάτων στον φάκελο καταχωρητών. Αυτή η τεχνική αν και απλή στην υλοποίηση της οδηγεί σε ένα μεγάλο κρίσιμο μονοπάτι (long critical path) αφού τα περισσότερα μέρη του επεξεργαστή συνδέονται σειριακά μεταξύ τους. Άρα από τον επεξεργαστή single-cycle δεν αναμένουμε μεγάλη απόδοση όσον αφορά την ταχύτητά του.

4.1 Βασικές Δομικές Μονάδες

Ένας λογικός τρόπος να ξεκινήσουμε την υλοποίηση του επεξεργαστή είναι να σχεδιάσουμε τη διαδρομή δεδομένων και να εξετάσουμε τις βασικές δομικές μονάδες που απαιτούνται για να εκτελεστεί κάθε μια από τις κατηγορίες των εντολών του MIPS που περιγράψαμε αναλυτικά στο Κεφ. 2.

Για την πλήρη εκτέλεση κάθε εντολής εκτελούνται πέντε βήματα. Τα δύο πρώτα βήματα είναι ίδια για όλες τις εντολές:

1. Αποστολή της διεύθυνσης που είναι αποθηκευμένη στον μετρητή προγράμματος (Program Counter - PC) στη μνήμη εντολών (PAMI) που περιέχει τον κώδικα του προγράμματος και προσκόμιση της εντολής στον επεξεργαστή.
2. Αποκωδικοποίηση της εντολής από τον controller και παραγωγή όλων των κατάλληλων σημάτων ελέγχου των πολυπλεκτών ή άλλων δομικών μονάδων. Κατά τη διάρκεια του ίδιου βήματος γίνεται η ανάγνωση ενός ή δύο καταχωρητών, με τη χρήση των πεδίων της εντολής για την επιλογή των καταχωρητών που θα διαβαστούν. Σημειώστε πως για την εντολή lw χρειάζεται να διαβάσουμε μόνο έναν καταχωρητή, αλλά οι περισσότερες από τις υπόλοιπες εντολές απαιτούν την ανάγνωση δύο καταχωρητών.

Μετά από αυτά τα δύο βήματα, οι ενέργειες που απαιτούνται για την ολοκλήρωση της εντολής εξαρτώνται από την κατηγορία που ανήκει. Για τις περισσότερες από τις εντολές, οι ενέργειες σε μεγάλο βαθμό είναι ίδιες, ανεξάρτητα από τον ακριβή κωδικό λειτουργίας. Ακολουθώντας λοιπόν τη διαδρομή των δεδομένων για κάθε εντολή διαπιστώνουμε την ανάγκη των παρακάτω δομικών μονάδων.

4.1.1 Φάκελος Καταχωρητών (Register File)

Ο φάκελος καταχωρητών είναι η τοπική αποθήκη του επεξεργαστή. Είναι η πιο μικρή μνήμη αλλά και η πιο κοντινή στη μονάδα επεξεργασίας (ALU). Οι περισσότερες λειτουργίες περιλαμβάνουν τη χρήση ή την τροποποίηση δεδομένων που βρίσκονται αποθηκευμένα στο φάκελο καταχωρητών. Επειδή ο φάκελος καταχωρητών προσπελάζεται σε κάθε εντολή πρέπει να είναι αρκετά γρήγορος ώστε να μην επιβαρύνει με καθυστερήσεις την ALU που περιμένει τους τελεστές για να εκτελέσει τις προαποφασισμένες εντολές.

Το entity του φακέλου καταχωρητών δίδεται παρακάτω:

```
entity registerFile is
      port
      (
            wdat        :     in    std_logic_vector (31 downto 0);
            wsel        :     in    std_logic_vector (4 downto 0);
            wen         :     in    std_logic;
            clk         :     in    std_logic;
            nReset      :     in    std_logic;
            rsel1       :     in    std_logic_vector (4 downto 0);
            rsel2       :     in    std_logic_vector (4 downto 0);
            rdat1       :     out   std_logic_vector (31 downto 0);
            rdat2       :     out   std_logic_vector (31 downto 0)
      );
   end registerFile;
```

Οι είσοδοι rsel1 και rsel2 είναι οι διευθύνσεις των δύο τελεστών οι οποίοι οδηγούνται τελικά στις πόρτες εξόδου rdat1 και rdat2 του φακέλου καταχωρητών, αντίστοιχα. Το wen είναι ένα bit επίτρεψης εγγραφής και τα wdat είναι τα δεδομένα που πρέπει να γραφούν στον καταχωρητή που δίδεται στη διεύθυνση wsel. Τέλος το clk και το nReset είναι βασικά σήματα για τη λειτουργία των καταχωρητών.

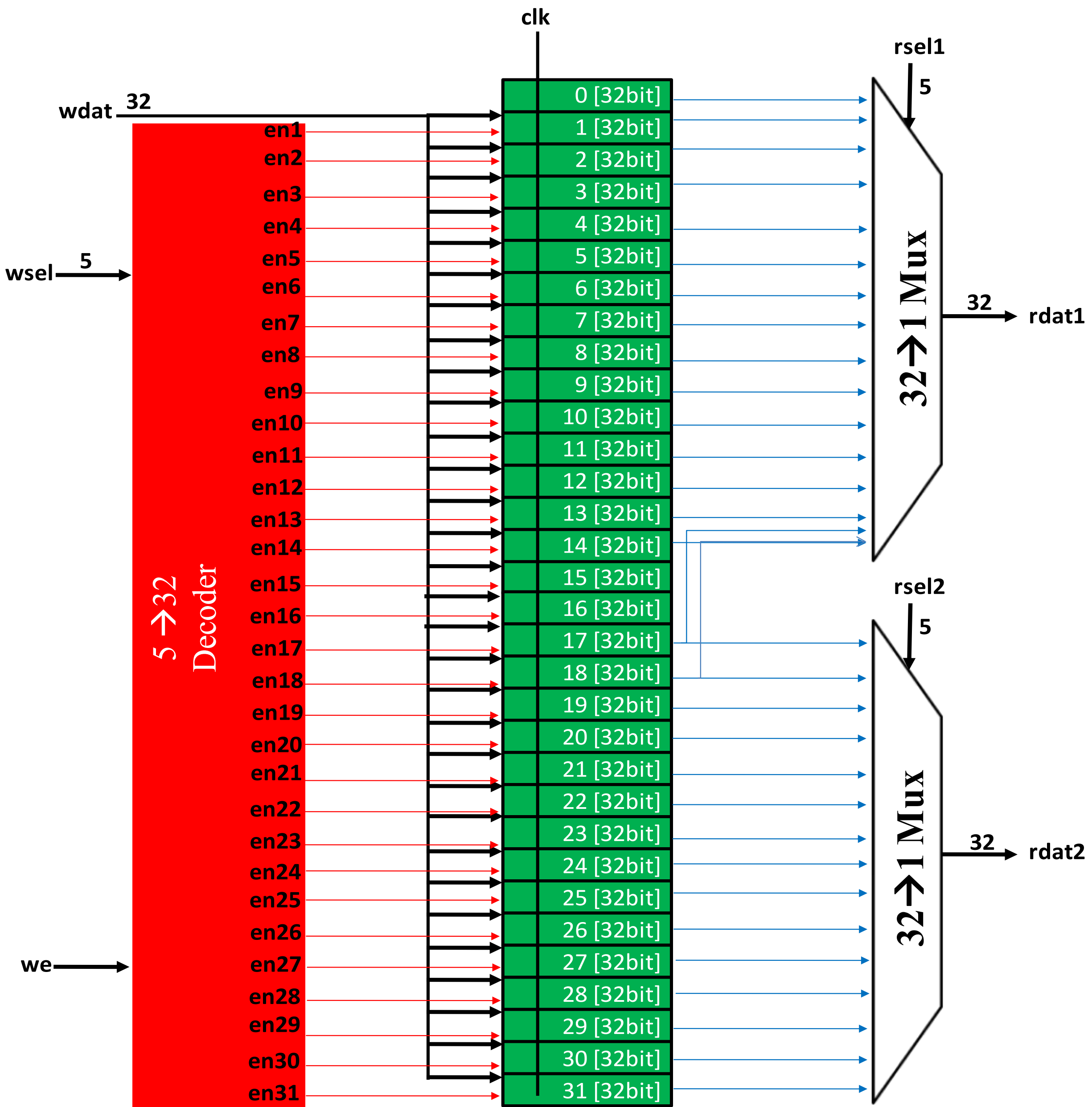
Στο φάκελο καταχωρητών που σχεδιάσαμε για τον επεξεργαστή MIPS, υπάρχουν 32 καταχωρητές, των 32 bits έκαστος. Κάθε καταχωρητής διαθέτει μία είσοδο ρολογιού (clk), μία είσοδο δεδομένων εγγραφής (wdat), ένα σήμα επίτρεψης εγγραφής (en) και μία έξοδο δεδομένων. Οι καταχωρητές αυτοί διαχειρίζονται από πολυπλέκτες και αποκωδικοποιητές όπως φαίνεται στο Σχ. 4.1-1. Υπάρχουν δύο έξοδοι από τον φάκελο καταχωρητών (read ports) έτσι ώστε σε κάθε κύκλο ρολογιού μπορούμε να διαβάσουμε δύο καταχωρητές, δηλαδή τα δύο τελούμενα. Όσον αφορά στα δεδομένα που μπορούμε να γράψουμε στον φάκελο καταχωρητών, κάθε φορά μας δίνεται η δυνατότητα μίας μόνο εισόδου δεδομένων (wdat), ενώ ποιού καταχωρητή τα δεδομένα θα ενημερωθούν επιλέγεται από το σήμα επιλογής (wsel). Αυτό βέβαια μπορεί να συμβεί μόνο όταν το wen έχει την τιμή ‘1’, οπότε στην άνοδο του ρολογιού γίνεται η ενημέρωση του καταχωρητή.

4.1.2 Αριθμητική Λογική Μονάδα (ALU)

Η αριθμητική λογική μονάδα (ALU) αποτελεί ένα από τα πιο σημαντικά μέρη του επεξεργαστή. Σε αυτή τη δομική μονάδα λαμβάνουν χώρα όλες οι μαθηματικές και λογικές πράξεις. Η ALU είναι καθαρά συνδυαστικό κύκλωμα και για την υλοποίηση της δεν χρησιμοποιούμε καθόλου καταχωρητές ή latches. Όπως θα δείξουμε από τη σύνθεση του κυκλώματος δεν χρησιμοποιήθηκε κανένα ακολουθιακό στοιχείο για την ALU.

Το entity της μονάδας ALU ακολουθεί:

```
entity alu is
      port
      (
            Opcode            :     in    std_logic_vector (2 downto 0);
            A, B              :     in    std_logic_vector (31 downto 0);
            Output            :     out   std_logic_vector (31 downto 0);
            Negative          :     out   std_logic;
            overflow, zero    :     out   std_logic
      );
end alu;
```



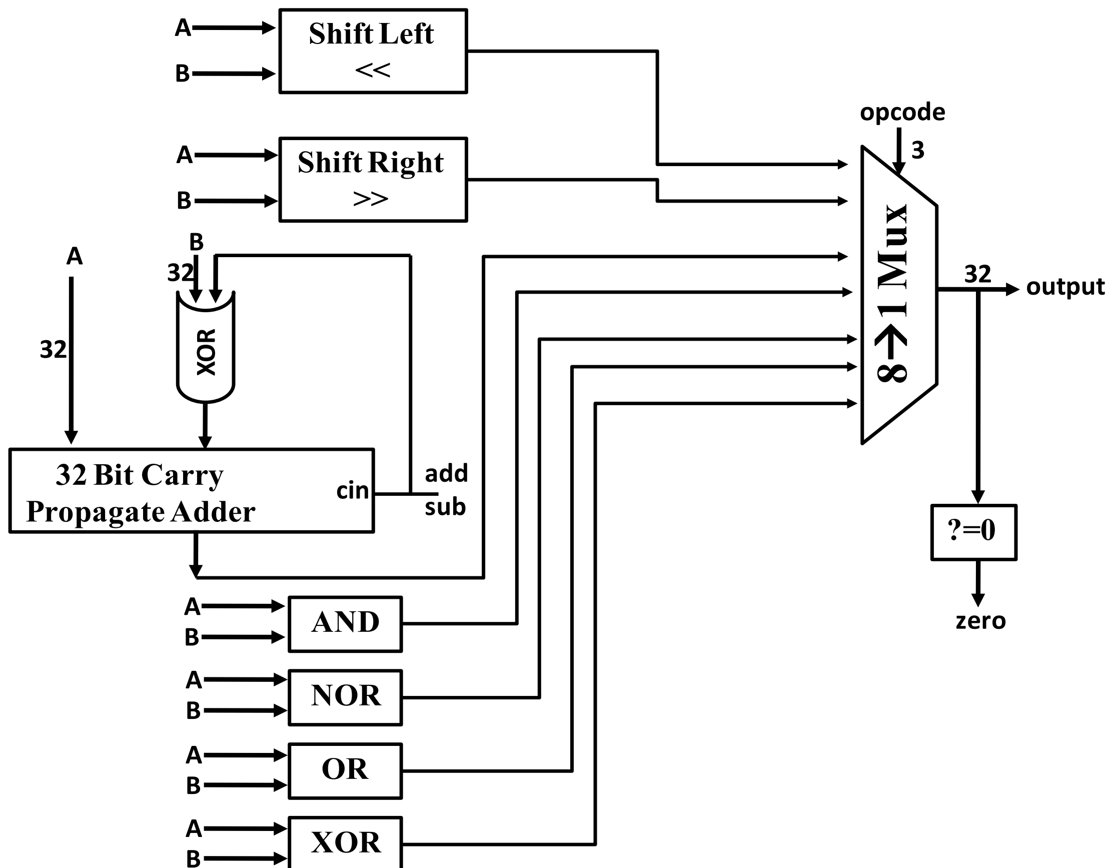
Σχ. 4.1-1: Σχηματικό διάγραμμα του φακέλου καταχωρητών. Αποτελείται από 32 καταχωρητές των 32bit ο καθένας. Είναι επίσης σχεδιασμένος ο decoder στην είσοδο του φακέλου καταχωρητών καθώς και οι δύο πολυπλέκτες στην έξοδό του.

Η επιλογή της λειτουργίας που θα εκτελέσει κάθε φορά η ALU γίνεται μέσω της εισόδου opcode η οποία καθορίζεται από την εντολή που αποκωδικοποιήθηκε. Οι πράξεις και τα αντίστοιχα opcodes που έχουμε υλοποιήσει για την ALU δίνονται στον Πίν. 4.1-1. Επίσης, τα δύο τελούμενα της πράξης που θα εκτελέσει η ALU δίνονται στις εισόδους A και B.

Τα σήμα εξόδου output έχει το αποτέλεσμα της πράξης της ALU. Το zero, overflow και negative είναι επίσης σήματα εξόδου. Πιο συγκεκριμένα το zero είναι ενεργό, δηλαδή ‘1’, μόνο εάν η έξοδος output της ALU είναι ίση με μηδέν. Το σήμα overflow είναι ενεργό, (‘1’) όταν μία μαθηματική πράξη οδηγήσει σε υπερχείλιση και το negative, αν το αποτέλεσμα είναι αρνητικό (μόνο για τις πράξεις της πρόσθεσης και της αφαίρεσης τα σήματα αυτά έχουν νόημα).

Opcode	Πράξη	Περιγραφή
000	**SLL**	OUTPUT<= A << B
001	**SRL**	OUTPUT<= A >> B
010	**ADD**	OUTPUT<= A + B
011	**SUB**	OUTPUT<= A – B
100	**AND**	OUTPUT<= A and B
101	**NOR**	OUTPUT<= A nor B
110	**OR**	OUTPUT<= A or B
111	**XOR**	OUTPUT<= A xor B

Πίν. 4.1-1: Πράξεις που έχουν υλοποιηθεί στην ALU και τα αντίστοιχα opcode που τις ενεργοποιούν.



Σχ. 4.1-2: Σχηματικό διάγραμμα του κυκλώματος της ALU (Arithmetic Logic Unit).

4.1.3 Μονάδα ελέγχου (Controller)

Η μονάδα ελέγχου αποκωδικοποιεί τις εντολές και προσδιορίζει τα πεδία της εντολής και τις γραμμές ελέγχου που χρειάζονται στους πολυπλέκτες του επεξεργαστή. Ο τρόπος αποκωδικοποίησης των εντολών διαφέρει ανάλογα με την κατηγορία της κάθε εντολής. Παρακάτω συνοψίζουμε σύντομα τα πεδία των τριών βασικών τύπων εντολών:

1. R-type

Opcode[31:26]	Rs[25:21]	Rt[20:16]	Rd[15:11]	Shamt[10:6]	Funct[5:0]

2. I-type

Opcode[31:26]	Rs[25:21]	Rt[20:16]	Immediate[15:0]

3. J-type

Opcode[31:26]	Address[25:0]

Το entity της μονάδας ελέγχου ακολουθεί:

```
entity controller is
      port
      (
            Instruction :     in    std_logic_vector (31 downto 0);
            Equal             :     in    std_logic;
            Rs,Rt,Rd          :     out   std_logic_vector (4 downto 0);
            Imm16             :     out   std_logic_vector (15 downto 0);
            ALUctr            :     out   std_logic_vector (3 downto 0);
            RegDst            :     out   std_logic_vector (1 downto 0);
            RegWr,MemWr :     out   std_logic;
            PCsel             :     out   std_logic_vector (1 downto 0);
            Jsel              :     out   std_logic;
            ExtOp             :     out   std_logic;
            ALUSrc            :     out   std_logic;
            MemtoReg          :     out   std_logic;
            LUIsel            :     out   std_logic;
            Halt              :     out   std_logic
      );
end controller;
```

Έτσι η μονάδα ελέγχου μεταφράζει τα κομμάτια της εντολής σε σήματα τα οποία καθορίζουν τις επιλογές πολυπλεκτών (π.χ. RegDst, ALUsrc, MemtoReg, LUIsel, Jsel, PCsel), της ALU (π.χ. ALUctr) για την πράξη που θα επιλέξουμε, καθώς και τα σήματα επίτρεψης για την εγγραφή της μνήμης δεδομένων και του Φακέλου Καταχωρητών (MemWr και RegWr) και το σήμα halt.

4.1.4 Μετρητής Εντολών (Program Counter)

Ο Μετρητής Εντολών περιέχει τη διεύθυνση της επόμενης προς εκτέλεση εντολής που βρίσκεται αποθηκευμένη στον καταχωρητή program counter. Για τη φυσιολογική ροή εκτέλεσης εντολών ο PC περιέχει την τιμή PC+4. Παρόλα αυτά, υπάρχουν εντολές οι οποίες αλλάζουν την εκτέλεσης ροής του προγράμματος από την κανονική ροή (σειριακή ροή). Αυτό συμβαίνει όταν αποκωδικοποιηθούν εντολές άλματος και διακλάδωσης οπότε ο καταχωρητής PC περιέχει τη διεύθυνση που καθορίζεται από την εκάστοτε εντολή άλματος.

Πιο συγκεκριμένα, για τις εντολές διακλάδωσης, η επόμενη τιμή του PC δίδεται συμβολικά από το {PC+4+SignEx(imm16),00}. Για τις εντολές άλματος j και jal η επόμενη τιμή του καταχωρητή PC είναι το {PC+4[31:26],INSTRUCTION[25:00],00}. Για εντολές άλματος από καταχωρητή (jr), η τιμή του καταχωρητή PC λαμβάνεται από τον φάκελο καταχωρητών.

Ανεξάρτητα από το πως καθορίζεται η επόμενη τιμή του καταχωρητή PC αυτή προωθείται στη μνήμη εντολών (RAMI) και έτσι επιλέγεται η κατάλληλη εντολή προς εκτέλεση.

Το entity του μετρητή εντολών ακολουθεί:

```
entity programCounter is
	port
	(
		CLK		:	in	std_logic;
		nReset		:	in	std_logic;
		PCWr		:	in	std_logic;
		PCsel		:	in	std_logic_vector(1 downto 0);
		Imm16		:	in	std_logic_vector(15 downto 0);
		JumpHigh	:	in	std_logic_vector(9 downto 0);
		RetAddr		:	in	std_logic_vector(31 downto 0);
		PC, PC_4	:	out	std_logic_vector(31 downto 0)
	);
end programCounter;
```

4.1.5 Μνήμη Εντολών & Δεδομένων & (Instruction & Data Cache)

Οι τελευταίες δομικές μονάδες που θα παρουσιάσουμε σε αυτή την ενότητα και που είναι απαραίτητες για την υλοποίηση του επεξεργαστή είναι οι μνήμες του. Οι μνήμες αυτές είναι προφανώς η μνήμη δεδομένων και εντολών. Σε αυτό το σημείο πρέπει να τονίσουμε την πρακτική δυσκολία που υπάρχει για την υλοποίηση των μνημών. Είναι προφανές πως μια μνήμη μπορεί να υλοποιηθεί με τη χρήση των flip-flops του FPGA. Δυστυχώς όμως το μέγεθός τους δεν μας επιτρέπει να ακολουθήσουμε αυτόν τον τρόπο υλοποίησης. Αυτό το πρόβλημα λύνεται με τη χρήση μνημών που παρέχονται από το FPGA ανεξάρτητα από τα flip-flops. Έτσι παρακάτω παραθέτουμε τα entity του rami και ramd όπως αυτά παράχθηκαν από τις βιβλιοθήκες του FPGA:

```
entity rami is
	port
	(
		Address		:	in	std_logic_vector (15 DOWNTO 0);
		clock		:	in	std_logic;
		data		:	in	std_logic_vector (31 DOWNTO 0);
		wren		:	in	std_logic;
		q		:	out	std_logic_vector (31 DOWNTO 0)
	);
end rami;


entity ramd is
	port
	(
		address		:	in	std_logic_vector (15 DOWNTO 0);
		clock		:	in	std_logic;
		data		:	in	std_logic_vector (31 DOWNTO 0);
		wren		:	in	std_logic;
		q		:	out	std_logic_vector (31 DOWNTO 0)
	);
end ramd;
```

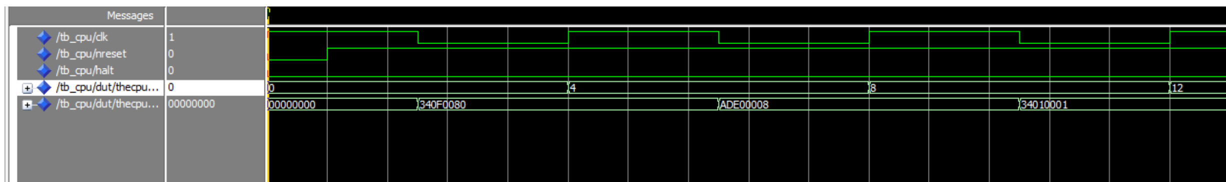
Παρατηρήστε ότι οι μνήμες έχουν είσοδο το ρολόι, τη διεύθυνση για ανάγνωση και εγγραφή, τα δεδομένα εισόδου για εγγραφή, ένα σήμα επίτρεψης της εγγραφής και μια έξοδο με τα δεδομένα που

διαβάστηκαν. Σημειώστε ότι η rami μας παρέχει είσοδο εγγραφής αλλά εμείς δεν θα τη χρησιμοποιήσουμε αφού η μνήμη εντολών μόνο διαβάζεται και δεν γράφεται ποτέ.

4.2 Προσομοίωση

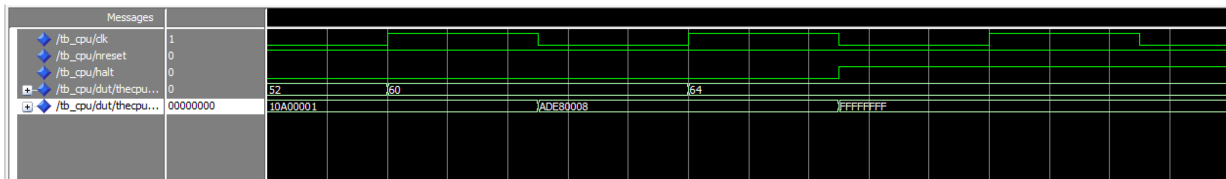
4.2.1 Προσομοίωση σε Λογικό επίπεδο

Εδώ κάνουμε την προσομοίωση σε λογικό επίπεδο του επεξεργαστή. Παρατηρούμε ότι ο επεξεργαστής γίνεται reset από 0 σε 1 αρχικά και έτσι ξεκινάει η εκτέλεση του προγράμματος που είναι αποθηκευμένο στην μνήμη εντολών. Επίσης ο μετρητής εντολών είναι αρχικοποιημένος στη διεύθυνση 0 που αντιστοιχεί στην πρώτη εντολή του προγράμματος. Παρατηρούμε ότι στην άνοδο του ρολογιού η μνήμη δέχεται τη διεύθυνση εντολών από τον PC και στην κάθοδο του ρολογιού επιστρέφει την αντίστοιχη εντολή. Τα παραπάνω σήματα φαίνονται στην Εικ. 4.2-3. Επίσης, σε κάθε κύκλο μηχανής μόνο μία εντολή διαβάζεται από τη μνήμη και μόνο μετά την ολοκλήρωσή της ξεκινάει η εκτέλεση της επόμενης. Αυτός είναι ο λόγος που ο συγκεκριμένος επεξεργαστής ονομάζεται Single-Cycle.



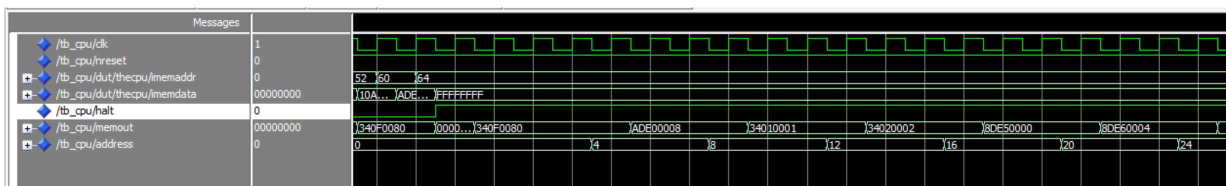
Εικ. 4.2-3: Αρχικοποίηση του επεξεργαστή με τη φόρτωση των εντολών του προγράμματος στη μνήμη εντολών.

Κατά τη διάρκεια της εκτέλεσης του προγράμματος το σήμα halt είναι στο λογικό 0. Όταν ο επεξεργαστής αποκωδικοποιήσει την εντολή τερματισμού FFFFFF κάνει το σήμα halt στο λογικό 1, όπως φαίνεται στην Εικ. 4.2-4.



Εικ. 4.2-1: Τερματισμός του προγράμματος από την εντολή FFFFFF και την ενεργοποίηση του σήματος τερματισμού halt.

Μετά τον τερματισμό του επεξεργαστή το testbench διαβάζει τα δεδομένα που της μνήμης RAM και σειριακά τα γράφει στο αρχείο εξόδου memout.hex, όπως περιγράψαμε στο Κεφ. 2. Στην Εικ. 4.2-5 βλέπουμε πως το testbench στέλνει τις κατάλληλες διευθύνσεις και η μνήμη επιστρέφει τα δεδομένα.



Εικ. 4.2-2: Μετά τον τερματισμό του προγράμματος το testbench ενεργοποιεί τη διαδικασία καταγραφής των δεδομένων της μνήμης σε αρχείο.

4.2.2 Σύνθεση και Χρονική Ανάλυση

Σε αυτή την υποενότητα περιγράφουμε τη διαδικασία κατά την οποία ο λειτουργικά ορθός κώδικας VHDL που περιγράφει τον επεξεργαστή single-cycle μετατρέπεται σε πραγματικό κύκλωμα που αποτελείται από λογικές πύλες και flip-flops. Όπως έχουμε αναφέρει όλα αυτά τα στοιχεία βρίσκονται στο core του FPGA. Έπειτα ακολουθεί η χρονική ανάλυση κατά την οποία καθορίζονται, ο μέγιστος χρόνος εκτέλεσης και κατά συνέπεια η ελάχιστη περίοδος ή η μέγιστη συχνότητα του ρολογιού.

Για τη σύνθεση του κυκλώματος χρησιμοποιήσαμε το πρόγραμμα Quartus II. Δημιουργήσαμε ένα project για κάθε επεξεργαστή και προσθέσαμε όλα τα απαραίτητα αρχεία VHDL. Σε αυτή την περίπτωση είναι προφανώς πως δεν εισάγαμε το testbench, γιατί δεν χρειάζεται κατά τη διαδικασία της σύνθεσης και γιατί δεν μπορεί να γίνει η σύνθεση του. Επίσης, κατά τη δημιουργία του project επιλέξαμε να χρησιμοποιηθούν οι βιβλιοθήκες που αντιστοιχούν στο FPGA Cyclone II της Altera.

Εικ. 4.2-3: Αποτελέσματα μετά από την επιτυχή μεταγλώττιση και σύνθεση του επεξεργαστή Single-Cycle.

Μετά το πέρας της εκτέλεσης της σύνθεσης του κυκλώματος λάβαμε την αναφορά του Quartus. Από την αναφορά αυτή (βλ. Εικ. 4.2-6) παρατηρούμε ότι ο επεξεργαστής single-cycle χρησιμοποιεί το 7% των λογικών στοιχείων του FPGA και καταλαμβάνει το 54% της συνολικής μνήμης του. Η μνήμη αυτή είναι η rami και ramd που περιγράφηκαν στις προηγούμενες υποενότητες. Επίσης, είναι αξιοσημείωτο ότι ο αριθμός των register που χρησιμοποιήθηκαν είναι 1160 ακριβώς όσους δηλαδή περιμέναμε όταν σχεδιάζαμε τον επεξεργαστή. Έτσι επιβεβαιώνουμε ότι δεν δημιουργήθηκαν επιπλέον ακολουθιακά στοιχεία (π.χ. flip-flops ή latches) που δημιουργήθηκαν κατά τη σύνθεση αλλά δεν περιγραφηκαν από τη VHDL.

Compilation Report - Timing Analyzer Summary | Timing Closure Floorplan

Compilation Report
Legal Notice
Flow Summary
Flow Settings
Flow Non-Default Global Se
Flow Elapsed Time
Flow Log
Analysis & Synthesis
Partition Merge
Fitter
Assembler
Timing Analyzer
Summary
Settings
Clock Settings Summary
Clock Setup: 'altera_inte
Clock Setup: 'CLK'
tsu
tco
tpd
th
Messages

Timing Analyzer Summary

	Type	Slack	Required Time	Actual Time	From
1	Worst-case tsu	N/A	None	7.988 ns	dumpAddr[7]
2	Worst-case tco	N/A	None	30.877 ns	mycpu:theCPU|rami:theRAM|altsyncram:altsyncr
3	Worst-case tpd	N/A	None	13.146 ns	dumpAddr[7]
4	Worst-case th	N/A	None	1.433 ns	altera_internal_jtag~TMSUTAP
5	Clock Setup: 'CLK'	N/A	None	16.63 MHz (period = 60.126 ns)	mycpu:theCPU|rami:theRAM|altsyncram:altsyncr
6	Clock Setup: 'altera_internal_jtag~TCKUTAP'	N/A	None	160.88 MHz (period = 6.216 ns)	mycpu:theCPU|rami:theRAM|altsyncram:altsyncr
7	Total number of failed paths				

Εικ. 4.2-4: Αποτελέσματα από την Ανάλυση Χρονισμού

Από την ανάλυση Χρονισμού παρατηρούμε ότι η μέγιστη ταχύτητα που μπορεί να τρέχει ο επεξεργαστής single-cycle για το συγκεκριμένο FPGA είναι 16MHz. Επίσης υπάρχει η δυνατότητα να δούμε πιο είναι το critical path που τελικά καθορίζει και τη μέγιστη συχνότητα του ρολογιού. Στην Εικ. 4.2-8 αναφέρονται τα 20 πιο κρίσιμα μονοπάτια. Παρατηρήστε ότι το πιο κρίσιμο μονοπάτι ξεκινάει από τη μνήμη εντολών και τερματίζει στον φάκελο καταχωρητών.

Στην Εικ. 4.2-9 βλέπουμε το floorplan όπως δίνεται από το Quartus. Με γαλάζιο χρώμα αναπαρίστανται οι περιοχές του FPGA που οποίες χρησιμοποιήθηκαν για την υλοποίηση του επεξεργαστή single-cycle. Σημειώστε ότι μόνο η μισή μνήμη του FPGA (μόνο η μισή μνήμη έχει γαλάζιο χρώμα) χρησιμοποιείτε για τις rami και ramd όπως ήδη διαπιστώσαμε και από τις παραπάνω αναφορές του Quartus. Τέλος, στην Εικ. 4.2-105 έχουμε κάνει μεγέθυνση σε μια μικρή περιοχή του floorplan που έχει υλοποιηθεί το λογικό μέρος του επεξεργαστή.

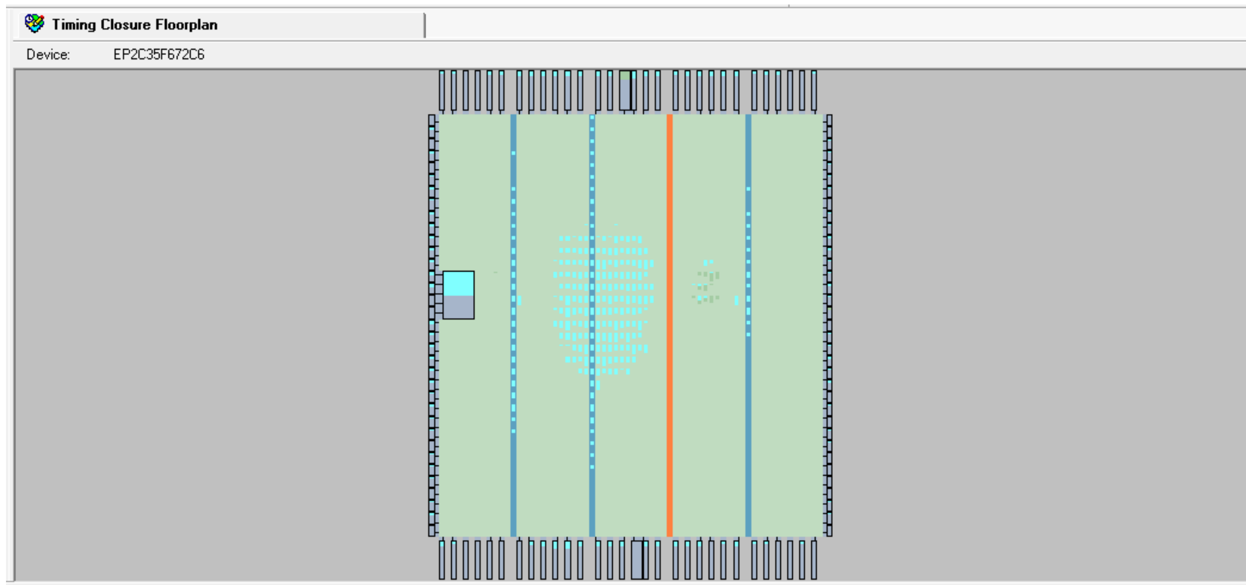
Compilation Report - Clock Setup: 'CLK' | Timing Closure Floorplan

Compilation Report
Legal Notice
Flow Summary
Flow Settings
Flow Non-Default Global Se
Flow Elapsed Time
Flow Log
Analysis & Synthesis
Partition Merge
Fitter
Assembler
Timing Analyzer
Summary
Settings
Clock Settings Summary
Clock Setup: 'altera_inte
Clock Setup: 'CLK'
tsu
tco
tpd
th
Messages

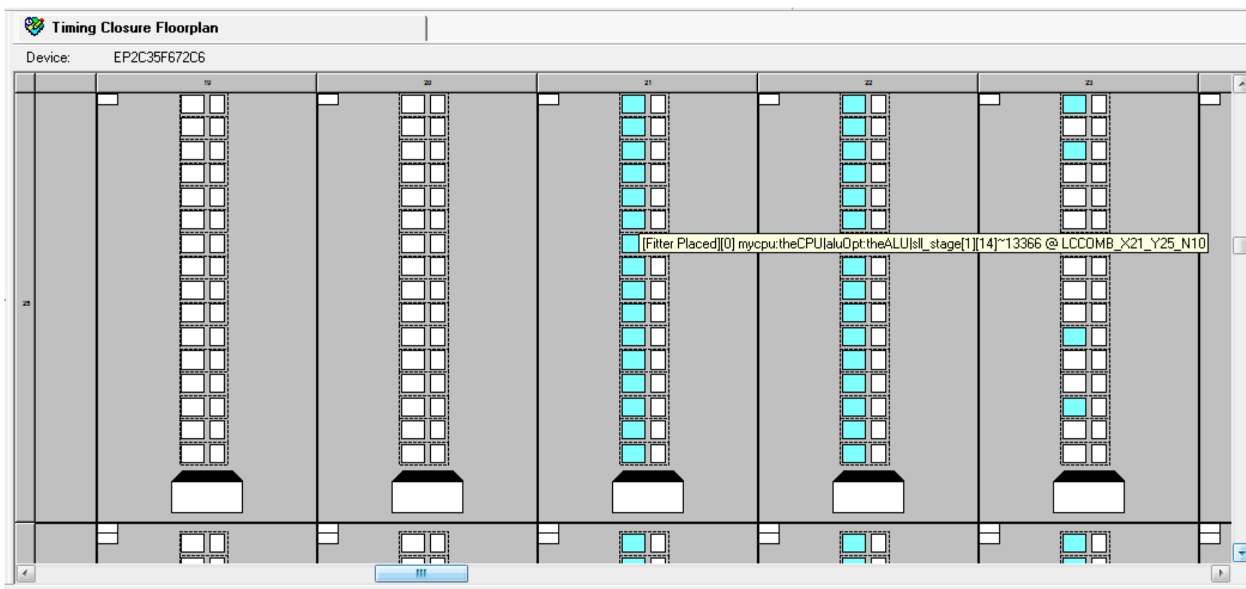
Clock Setup: 'CLK'

	Slack	Actual fmax (period)	From	To	From Clock	To Clock	Required Setup Relationship	Required Longest P2P Time
1	N/A	16.63 MHz (period = 60.126 ns)	mycpu:theCPU|rami:theRAM|a...	mycpu:theCPU|registerFile:theRegFile|reg[14][29]	CLK	CLK	None	None
2	N/A	16.63 MHz (period = 60.126 ns)	mycpu:theCPU|rami:theRAM|a...	mycpu:theCPU|registerFile:theRegFile|reg[14][29]	CLK	CLK	None	None
3	N/A	16.63 MHz (period = 60.126 ns)	mycpu:theCPU|rami:theRAM|a...	mycpu:theCPU|registerFile:theRegFile|reg[14][29]	CLK	CLK	None	None
4	N/A	16.63 MHz (period = 60.126 ns)	mycpu:theCPU|rami:theRAM|a...	mycpu:theCPU|registerFile:theRegFile|reg[14][29]	CLK	CLK	None	None
5	N/A	16.63 MHz (period = 60.126 ns)	mycpu:theCPU|rami:theRAM|a...	mycpu:theCPU|registerFile:theRegFile|reg[14][29]	CLK	CLK	None	None
6	N/A	16.63 MHz (period = 60.126 ns)	mycpu:theCPU|rami:theRAM|a...	mycpu:theCPU|registerFile:theRegFile|reg[14][29]	CLK	CLK	None	None
7	N/A	16.63 MHz (period = 60.126 ns)	mycpu:theCPU|rami:theRAM|a...	mycpu:theCPU|registerFile:theRegFile|reg[14][29]	CLK	CLK	None	None
8	N/A	16.63 MHz (period = 60.126 ns)	mycpu:theCPU|rami:theRAM|a...	mycpu:theCPU|registerFile:theRegFile|reg[14][29]	CLK	CLK	None	None
9	N/A	16.63 MHz (period = 60.126 ns)	mycpu:theCPU|rami:theRAM|a...	mycpu:theCPU|registerFile:theRegFile|reg[14][29]	CLK	CLK	None	None
10	N/A	16.63 MHz (period = 60.126 ns)	mycpu:theCPU|rami:theRAM|a...	mycpu:theCPU|registerFile:theRegFile|reg[14][29]	CLK	CLK	None	None
11	N/A	16.63 MHz (period = 60.126 ns)	mycpu:theCPU|rami:theRAM|a...	mycpu:theCPU|registerFile:theRegFile|reg[14][29]	CLK	CLK	None	None
12	N/A	16.63 MHz (period = 60.126 ns)	mycpu:theCPU|rami:theRAM|a...	mycpu:theCPU|registerFile:theRegFile|reg[14][29]	CLK	CLK	None	None
13	N/A	16.73 MHz (period = 59.770 ns)	mycpu:theCPU|rami:theRAM|a...	mycpu:theCPU|registerFile:theRegFile|reg[14][29]	CLK	CLK	None	None
14	N/A	16.73 MHz (period = 59.770 ns)	mycpu:theCPU|rami:theRAM|a...	mycpu:theCPU|registerFile:theRegFile|reg[14][29]	CLK	CLK	None	None
15	N/A	16.73 MHz (period = 59.770 ns)	mycpu:theCPU|rami:theRAM|a...	mycpu:theCPU|registerFile:theRegFile|reg[14][29]	CLK	CLK	None	None
16	N/A	16.73 MHz (period = 59.770 ns)	mycpu:theCPU|rami:theRAM|a...	mycpu:theCPU|registerFile:theRegFile|reg[14][29]	CLK	CLK	None	None
17	N/A	16.73 MHz (period = 59.770 ns)	mycpu:theCPU|rami:theRAM|a...	mycpu:theCPU|registerFile:theRegFile|reg[14][29]	CLK	CLK	None	None
18	N/A	16.73 MHz (period = 59.770 ns)	mycpu:theCPU|rami:theRAM|a...	mycpu:theCPU|registerFile:theRegFile|reg[14][29]	CLK	CLK	None	None
19	N/A	16.73 MHz (period = 59.770 ns)	mycpu:theCPU|rami:theRAM|a...	mycpu:theCPU|registerFile:theRegFile|reg[14][29]	CLK	CLK	None	None
20	N/A	16.73 MHz (period = 59.770 ns)	mycpu:theCPU|rami:theRAM|a...	mycpu:theCPU|registerFile:theRegFile|reg[14][29]	CLK	CLK	None	None
21	N/A	16.73 MHz (period = 59.770 ns)	mycpu:theCPU|rami:theRAM|a...	mycpu:theCPU|registerFile:theRegFile|reg[14][29]	CLK	CLK	None	None

Εικ. 4.2-8: Τα 20 πρώτα πιο μακριά κρίσιμα μονοπάτια. Το πιο κρίσιμο μονοπάτι καθορίζει τη μέγιστη συχνότητα του επεξεργαστή που σε αυτή την περίπτωση είναι 16MHz.

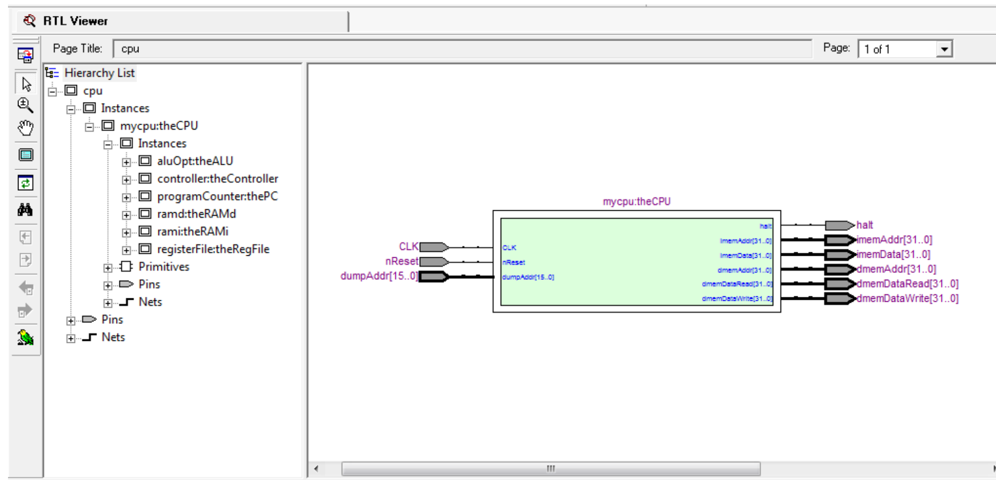


Εικ. 4.2-9: Γενική κάτοψη του floorplan.

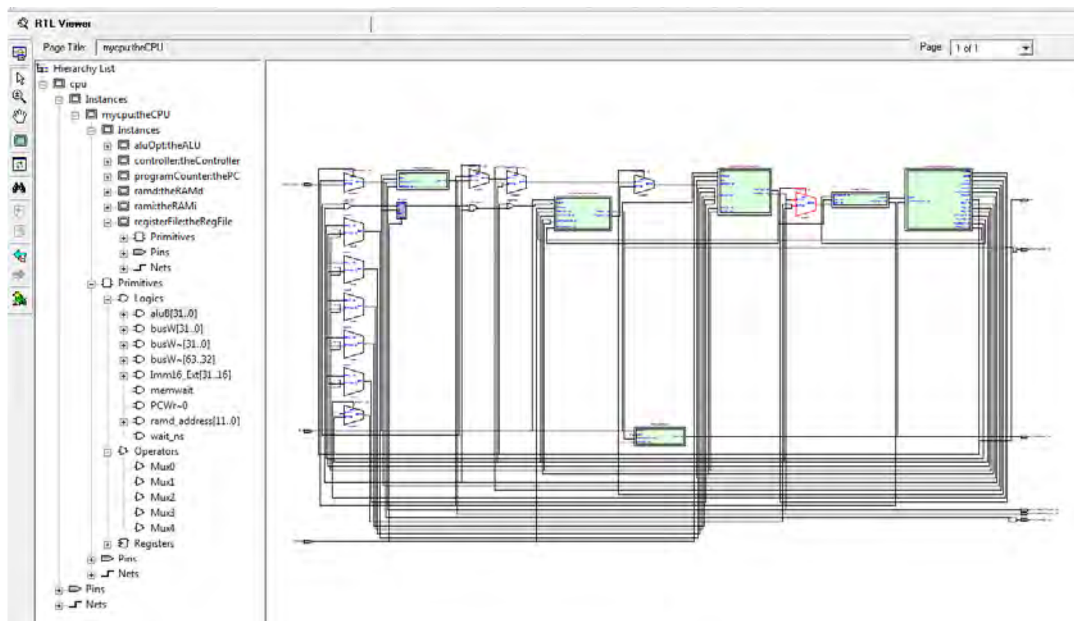


Εικ. 4.2-60: Τμήμα του floorplan (σε μεγέθυνση)

Στις Εικ. 4.2-10 και Εικ. 4.2-11 φαίνονται τα διαγράμματα όλου του επεξεργαστή αλλά και τμήμα αυτού σε μεγέθυνση. Είναι φανερό ότι ο επεξεργαστής έχει εισόδους το ρολόι και το reset και έξοδο το σήμα halt που ενημερώνει ότι το προς εκτέλεση πρόγραμμα τερμάτισε. Παρατηρήστε ότι υπάρχουν και κάποιοι άλλοι είσοδοι-έξοδοι οι οποίες όμως δεν είναι σημαντικές στη λειτουργία του επεξεργαστή. Ο λόγος ύπαρξής τους είναι για debugging του τελευταίου.



Εικ. 4.2-10: Σχηματικό διάγραμμα όπως αυτό προκύπτει έπειτα από τη σύνθεση του επεξεργαστή.



Εικ. 4.2-11: Σχηματικό διάγραμμα σε μεγέθυνση.

4.3 Συμπεράσματα

Σε αυτό το πρώτο μέρος της εργασίας υλοποιήσαμε επιτυχώς τον επεξεργαστή single-cycle. Επιβεβαιώσαμε την ορθή λειτουργία του τόσο σε λογικό (ModelSim) όσο και σε πραγματικό περιβάλλον (FPGA). Επίσης, μετρήσαμε την απόδοσή του τρέχοντας τη χρονική ανάλυση. Με την ολοκλήρωση αυτού του μέρους είμαστε σίγουροι για την ορθή λειτουργία αρκετών δομικών μονάδων που θα χρησιμοποιηθούν στις επόμενες υλοποιήσεις. Τέλος, έγινε εξοικείωση με τα βασικά εργαλεία, προγράμματα που χρησιμοποιήθηκαν σε αυτή τη διπλωματική εργασία, καθώς και με τη μεθοδολογία που περιγράφεται από το flowchart του Σχ. 2.1-1:Γενικό σχηματικό διάγραμμα της επικοινωνίας των cores ενός επεξεργαστή dual-core. Το κοινό τμήμα των cores είναι η μνήμη RAM και η επικοινωνία γίνεται μέσω του bus.Σχ. 2.1-1.

5. Pipeline processor

Το pipeline είναι μία τεχνική υλοποίησης σε επίπεδο αρχιτεκτονικής, σύμφωνα με την οποία η εκτέλεση διαδοχικών εντολών γίνεται σε επικάλυψη. Χρησιμοποιώντας αυτήν την τεχνική αναμένουμε ο επεξεργαστής MIPS να είναι αρκετά πιο γρήγορος. Αυτό συμβαίνει επειδή πολλές εντολές εκτελούνται παράλληλα οπότε ανά χρονική στιγμή εκτελούνται περισσότερες εντολές. Σημειώνουμε ότι το pipeline βελτιώνει την ικανότητα διεκπεραίωσης (throughput) του επεξεργαστή και όχι το χρόνο ολοκλήρωσης μιας συγκεκριμένης εντολής, ο οποίος είναι σχεδόν ίδιος με αυτόν του Single-cycle.

Ο επεξεργαστής MIPS που υλοποιούμε σε αυτήν την εργασία, απαιτεί 5 στάδια για την διεκπεραίωση της κάθε εντολής. Τα στάδια αυτά είναι:

1. Προσκόμιση (fetch) της εντολής από τη μνήμη εντολών (instruction cache).
2. Αποκωδικοποίηση της εντολής και ανάγνωση από τον φάκελο καταχωρητών (register file).
3. Εκτέλεση λογικής ή αριθμητικής πράξης ή υπολογισμός τελικής διεύθυνσης.
4. Προσπέλαση της μνήμης δεδομένων (data cache), εφόσον αυτό απαιτείται από την εντολή.
5. Εγγραφή του αποτελέσματος σε έναν καταχωρητή.

Στην περίπτωση της διοχέτευσης υπάρχουν εντολές που δε μπορούν να ξεκινήσουν την εκτέλεσή τους στον επόμενο κύκλο ρολογιού. Αυτές οι περιπτώσεις ονομάζονται κίνδυνοι (hazards) και υπάρχουν τρεις διαφορετικοί τύποι: δομικοί κίνδυνοι, κίνδυνοι δεδομένων και κίνδυνοι ελέγχου. Όπως θα δείξουμε παρακάτω, οι δομικοί κίνδυνοι δημιουργούνται λόγω της κοινής μνήμης εντολών και δεδομένων και αντιμετωπίζονται με τη χρήση ενός διαχειριστή (arbitrator). Οι κίνδυνοι δεδομένων εμφανίζονται όταν μία εντολή αναμένει δεδομένα από μία άλλη εντολή που είναι υπό εκτέλεση. Η αντιμετώπιση αυτών των κινδύνων γίνεται με υλοποίηση μίας μονάδας προώθησης (forward). Τέλος οι κίνδυνοι ελέγχου υπάρχουν όταν ο επεξεργαστή θέλει να εκτελέσει εντολές διακλάδωσης και αντιμετωπίζονται με την τοποθέτηση μίας κενής εντολής (NOP).

5.1 Βασικές Δομικές Μονάδες

Στην υλοποίηση του pipeline επεξεργαστή εκτός από τις μονάδες που έχουμε περιγράψει στην ενότητα 4.1, υλοποιούνται επιπλέον οι υπομονάδες των cache εντολών και δεδομένων και ένας arbitrator ο οποίος είναι υπεύθυνος για τη διαχείριση της πρόσβασης στη RAM.

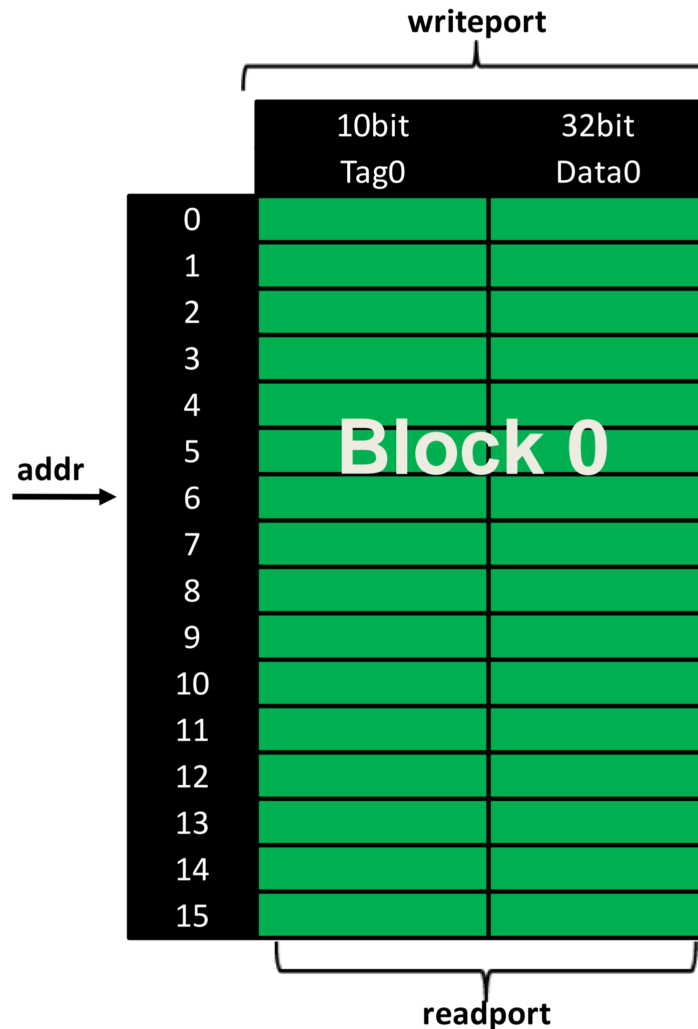
5.1.1 Μνήμη Εντολών (Instruction Cache)

Το entity της μνήμης εντολών δίδεται παρακάτω. Έχει σήματα εισόδου το clk και το nReset επειδή η icache είναι μια μνήμη. Επικοινωνεί από τη μία πλευρά με το pipeline και από την άλλη με τον arbitrator. Η επικοινωνία με το pipeline επιτυγχάνεται μέσω των σημάτων: `iMemRead`, `iMemAddr`, `iMemWait` και `iMemData` ενώ η επικοινωνία με τον arbitrator επιτυγχάνεται με τα σήματα: `aiMemWait`, `aiMemData`, `aiMemRead` και `aiMemAddr`.

```
entity icache is
  port(
    clk       : in  std_logic;
    nReset    : in  std_logic;

    iMemRead  : in  std_logic;                          -- CPU side
    iMemAddr  : in  std_logic_vector (31 downto 0);    -- CPU side
    iMemWait  : out std_logic;                          -- CPU side
    iMemData  : out std_logic_vector (31 downto 0);    -- CPU side
```

```
    aiMemWait : in  std_logic;                          -- arbitrator side
    aiMemData : in  std_logic_vector (31 downto 0);     -- arbitrator side
    aiMemRead : out std_logic;                          -- arbitrator side
    aiMemAddr : out std_logic_vector (31 downto 0)      -- arbitrator side
    );
end icache;
```

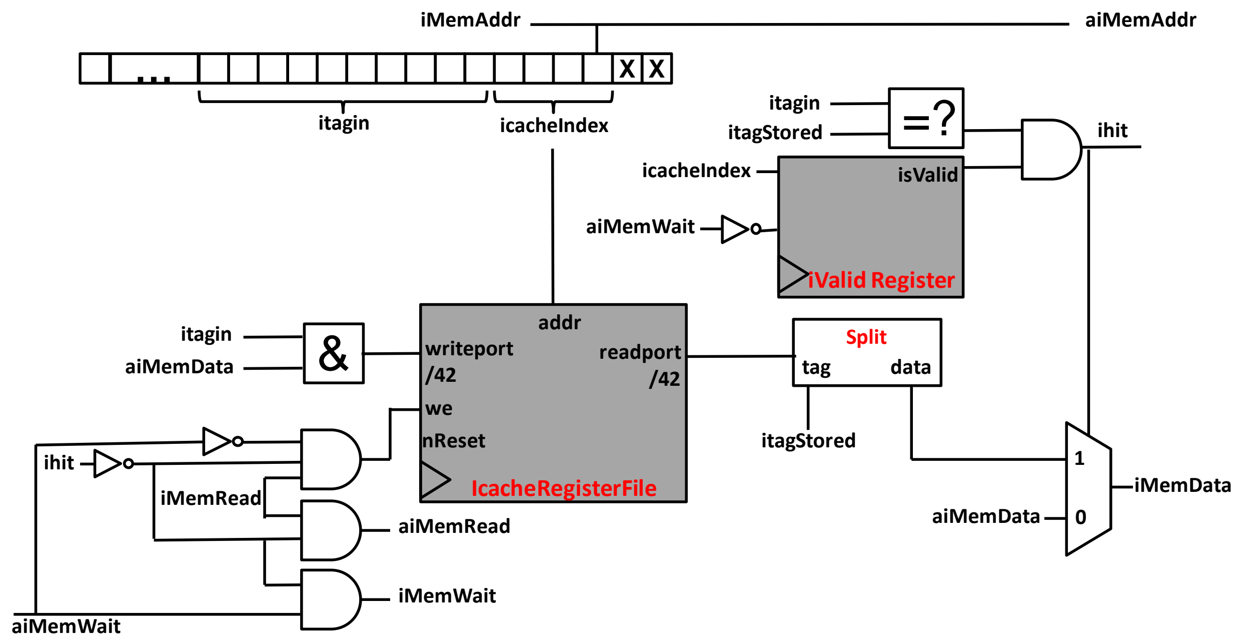
Το σχηματικό διάγραμμα της cache εντολών δίδεται στο Σχ. 5.1-1 . Η cache εντολών αποτελείται από το IcacheRegisterFile που είναι ένας πίνακας-μνήμη, όπου αποθηκεύονται οι εντολές και τα αντίστοιχα tags. Η cache εντολών αποτελείται από 16 block lines (γι αυτό και το addr είναι 4 bits) και είναι άμεσης συσχέτισης (direct associatively).



Σχ. 5.1-1: Σχηματικό διάγραμμα της icache.

Μια από τις πιο βασικές εισόδους αυτού του block είναι το `iMemAddr` δηλαδή η διεύθυνση που δίδεται στην icache από τον μετρητή εντολών. Η ίδια διεύθυνση διοχετεύεται άμεσα στον arbitrator μέσω του `aiMemAddr` έτσι ώστε ο arbitrator να επιστρέψει τα δεδομένα από τη RAM σε περίπτωση ενός cache miss. Το `iMemAddr` διασπάται σε 10 bit tag (itagIn) και 4 bit index τα οποία χρησιμοποιούνται ως διεύθυνση για το `IcacheRegisterFile`. Σαν αποτέλεσμα το `IcacheRegisterFile` θα επιστρέψει τα δεδομένα 42bit στην πόρτα εξόδου (readport). Τα

δεδομένα αυτά διασπώνται α. στα δεδομένα (σε αυτή την περίπτωση μια εντολή εφόσον αυτή αποδειχθεί έγκυρη) και β. στο `itagStored`. Το `itagStored` και το `itagin` εξετάζονται αν είναι ίσα και ταυτόχρονα αν το αντίστοιχο bit είναι έγκυρο. Αν και οι δύο αυτές συνθήκες είναι αληθείς τότε έχουμε hit (το bit `ihit` γίνεται ένα) αλλιώς έχουμε icache miss (το bit `ihit` γίνεται μηδέν) και θα πρέπει να διαβάσουμε έγκυρα δεδομένα από τη RAM.



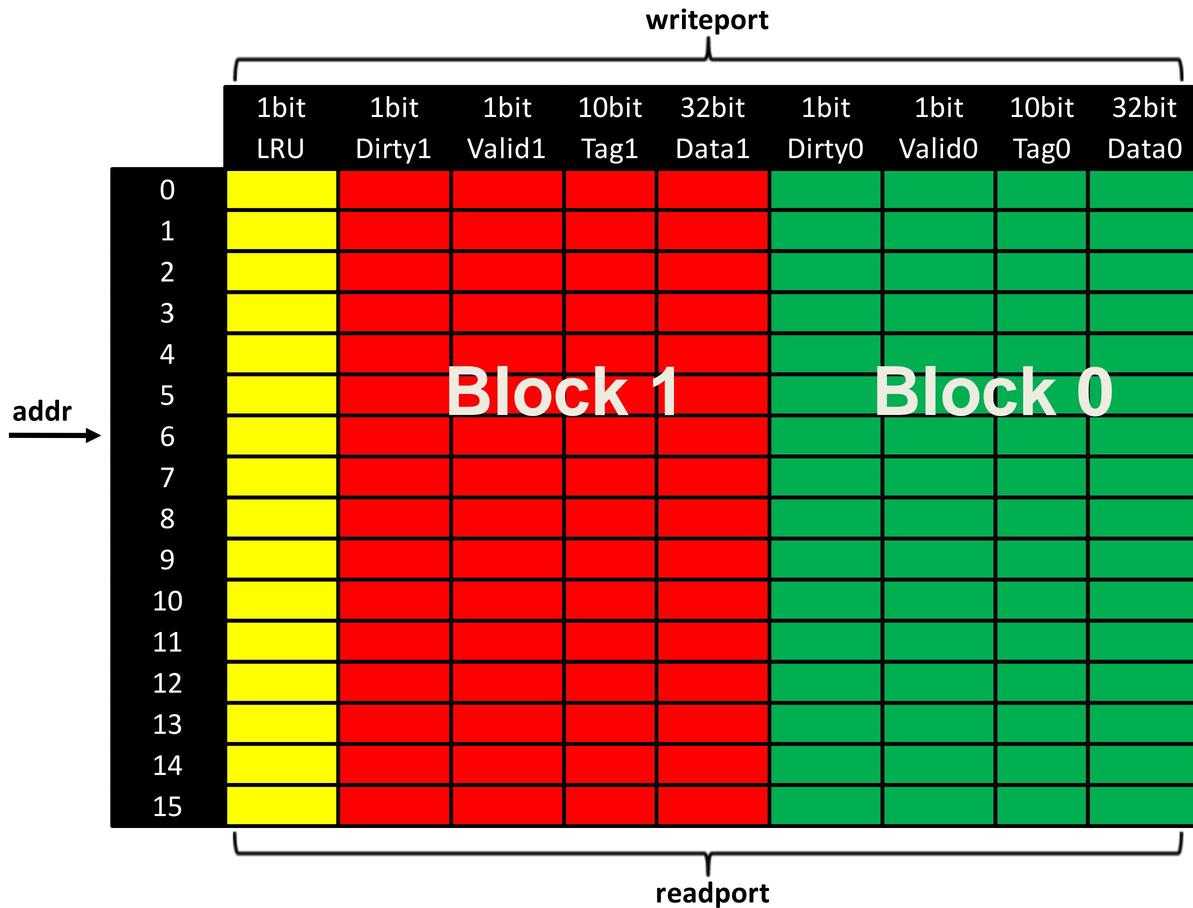
Σχ. 5.1-2: Μνήμη εντολών.

Όπως έχουμε ήδη προαναφέρει λόγω της περίπτωσης δομικού κινδύνου στη RAM τόσο η icache όσο και η dcache δεν επικοινωνούν άμεσα με τη RAM αλλά μέσω του arbitrator ο οποίος δίνει τις κατάλληλες προτεραιότητες. Σε αυτό το σημείο όμως σε περίπτωση miss η icache θα πρέπει να ενημερώσει τον arbitrator ότι χρειάζεται δεδομένα από τη RAM. Αυτό επιτυγχάνεται μέσω του σήματος aiMemRead το οποίο όταν ο επεξεργαστής έχει ζητήσει μια νέα εντολή και η icache δεν μπορεί να την παρέχει (miss). Σε πιο σημείο της RAM θα γίνει η αναζήτηση αυτό το έχουμε ήδη δώσει στον arbitrator μέσω του σήματος aiMemAddr που είναι το ίδιο με το iMemAddr. Αν ο arbitrator δεν μπορεί να δώσει πρόσβαση στη RAM επειδή εκείνη τη χρονική στιγμή τη χρησιμοποιεί η dcache τότε ο arbitrator ενημερώνει την icache να περιμένει, μέσω του aiMemWait. Με τη σειρά της η icache ενημερώνει το pipeline του επεξεργαστή ότι η επόμενη εντολή δεν είναι έτοιμη και πρέπει να περιμένει μέχρι να έρθει από τη RAM. Η ενημέρωση του pipeline γίνεται μέσω του σήματος iMemWait.

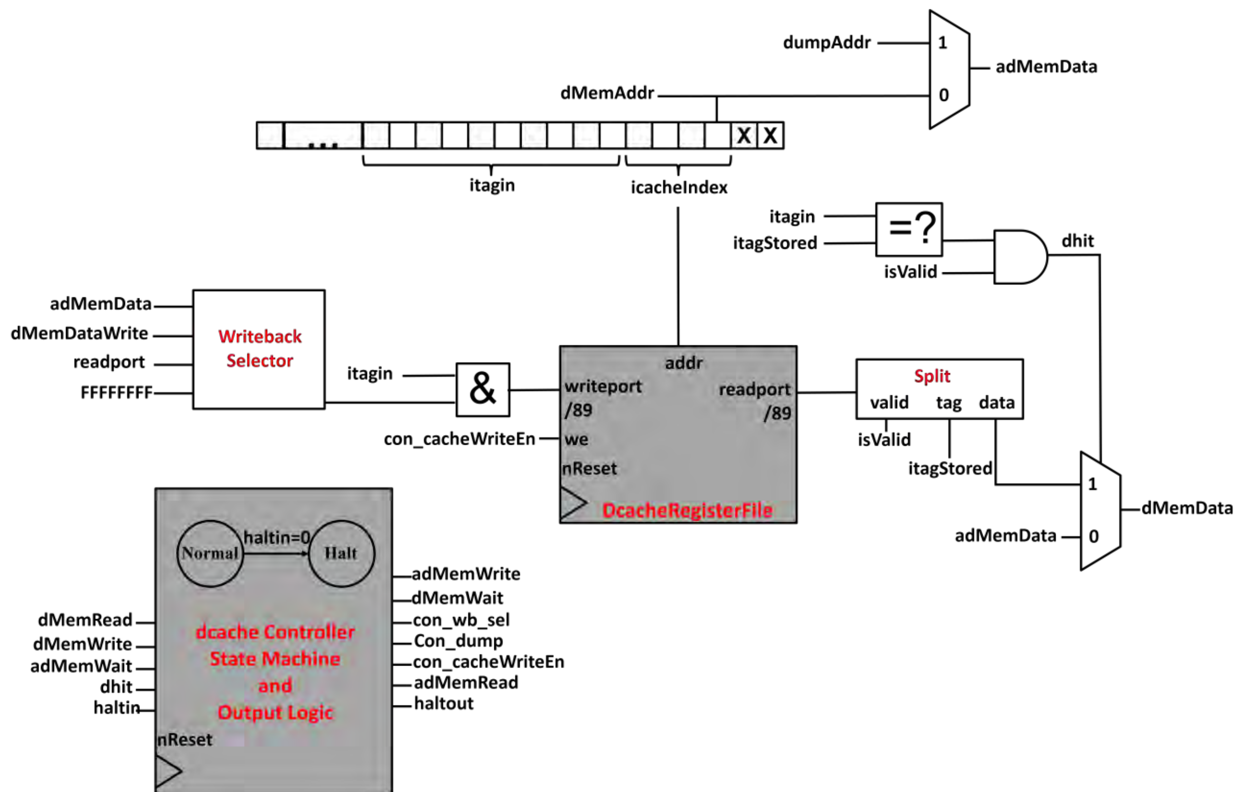
5.1.2 Μνήμη Δεδομένων (Data Cache)

```
entity dcache is
  port(
    clk             : in  std_logic;
    nReset          : in  std_logic;

    dMemRead        : in  std_logic;                       -- CPU side
    dMemWrite       : in  std_logic;                       -- CPU side
    dMemAddr        : in  std_logic_vector (31 downto 0);  -- CPU side
```

```
    dMemDataWrite  : in  std_logic_vector (31 downto 0);  -- CPU side
    dMemWait       : out std_logic;                       -- CPU side
    dMemDataRead   : out std_logic_vector (31 downto 0);  -- CPU side

    adMemWait      : in  std_logic;                       -- arbitrator side
    adMemDataRead  : in  std_logic_vector (31 downto 0);  -- arbitrator side
    adMemRead      : out std_logic;                       -- arbitrator side
    adMemWrite     : out std_logic;                       -- arbitrator side
    adMemAddr      : out std_logic_vector (31 downto 0);  -- arbitrator side
    adMemDataWrite : out std_logic_vector (31 downto 0);  -- arbitrator side

    haltin : in std_logic;--needed so that the cache is dumped before halting
    haltout        : out std_logic;
    tsl            : in std_logic   -- high when a tsl instruction is used

    );

end dcache;
```



Σχ. 5.1-3: Σχηματικό διάγραμμα της μνήμης δεδομένων (dcacheRegisterFile) για τον επεξεργαστή pipeline. Η dcache έχει δύο blocks σε αντιθεση με την icache, δηλαδή είναι 2-δρόμων συσχέτησης.



Σχ. 5.1-4: Συνολικό σχηματικό διάγραμμα της μνήμης δεδομένων (dcache) για τον επεξεργαστή pipeline.

5.1.3 Διαχειριστής Μνήμης (Arbitrator)

Ο arbitrator είναι ένας μεσολαβητής μεταξύ της icache και της dcache και ο σκοπός του είναι να λύσει το δομικό κίνδυνο που υπάρχει λόγω της ύπαρξης μια μόνο RAM. Το entity του arbitrator δίδεται παρακάτω:

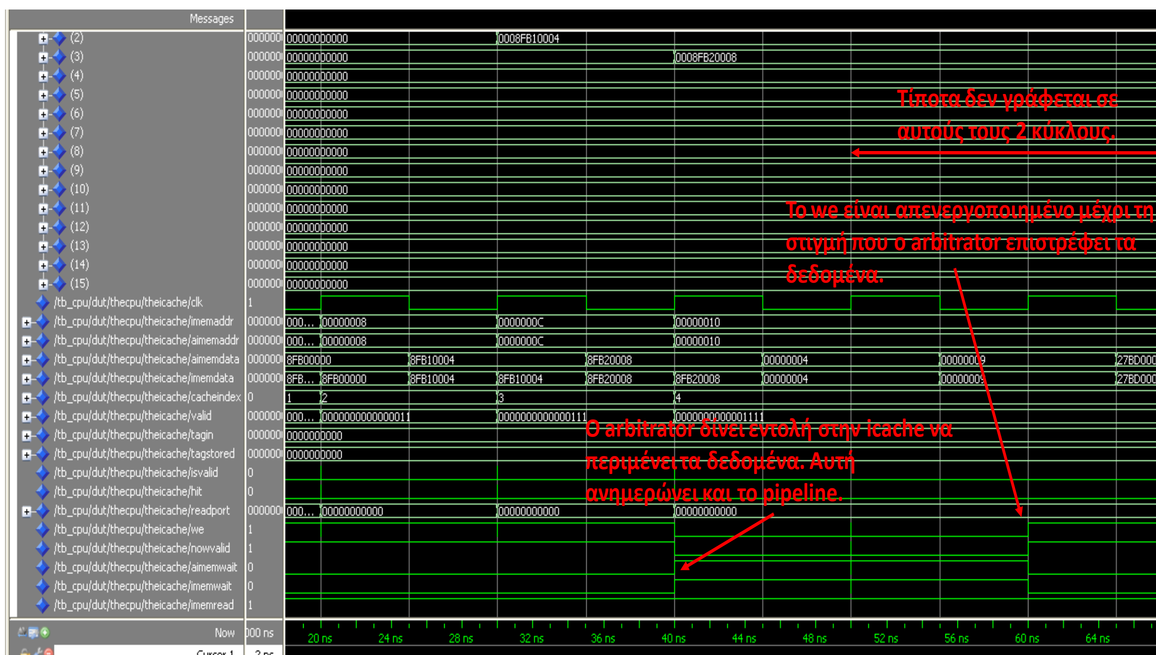
```
entity arbitrator is
  port(
    clk             : in  std_logic;
    nReset          : in  std_logic;

    -- instruction cache
    aiMemRead : in std_logic;
    aiMemAddr : in std_logic_vector (31 downto 0);
    aiMemWait : out  std_logic;
    aiMemData : out  std_logic_vector (31 downto 0);

    -- data cache
    adMemRead       : in std_logic;
    adMemWrite      : in std_logic;
    adMemAddr       : in std_logic_vector (31 downto 0);
    adMemDataWrite  : in std_logic_vector (31 downto 0);
    adMemWait       : out  std_logic;
    adMemDataRead   : out  std_logic_vector (31 downto 0);
```

```
    -- external RAM
    ramWr           : out std_logic;
    ramAddr         : out std_logic_vector(15 downto 0);
    ramWrite        : out std_logic_vector(31 downto 0);
    ramRead         : in std_logic_vector(31 downto 0)
    );

end arbitrator;
```

5.2 Προσομοίωση

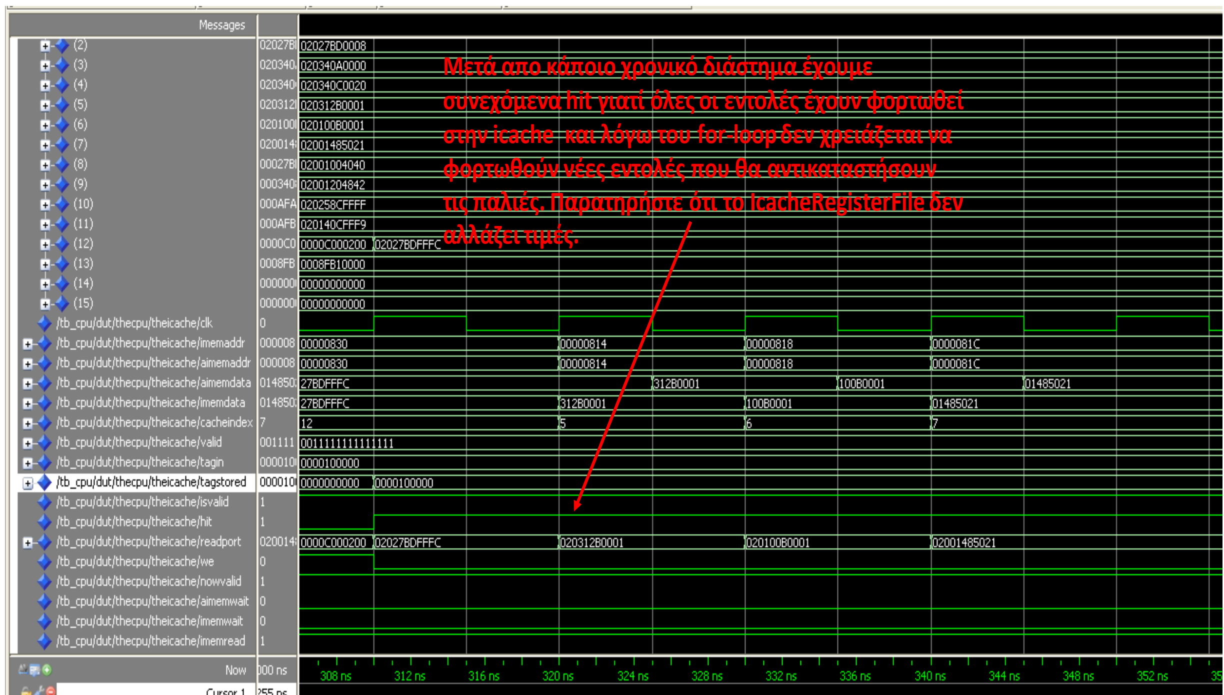
5.2.1 Προσομοίωση σε λογικό επίπεδο

Ας δούμε με μεγαλύτερη λεπτομέρεια πως ακριβώς δουλεύει η icache και πώς επικοινωνεί σε αρμονία με τον arbitrator για να δίνουν τις εντολές που έχουν ζητηθεί από το pipeline. Αρχικά από τον PC ζητείται η εντολή στη διεύθυνση 0 η οποία αρχικά δεν υπάρχει στην icache (αφού μετά από το reset όλα τα blocks είναι μη έγκυρα). Η διεύθυνση του PC, που πλέον είναι αποθηκευμένη στο σήμα iMemAddr, έχει προωθηθεί και στον arbitrator ο οποίος στο κατέβασμα του ρολογιού επιστρέφει την εντολή που βρίσκεται στη θέση 0 της RAM. Στο ανέβασμα του ρολογιού τα δεδομένα γράφονται στο IcacheRegisterFile και ταυτόχρονα τα λαμβάνει και ο decoder. Σημειώστε ότι στο ίδιο ανέβασμα του ρολογιού ο PC δίνει στην icache μια νέα διεύθυνση για τη φόρτωση της επόμενης εντολής. Επίσης το bit εγκυρότητας για την εντολή 0, που σύμφωνα με το index της αποθηκεύτηκε στην πρώτη γραμμή, γίνεται 1.



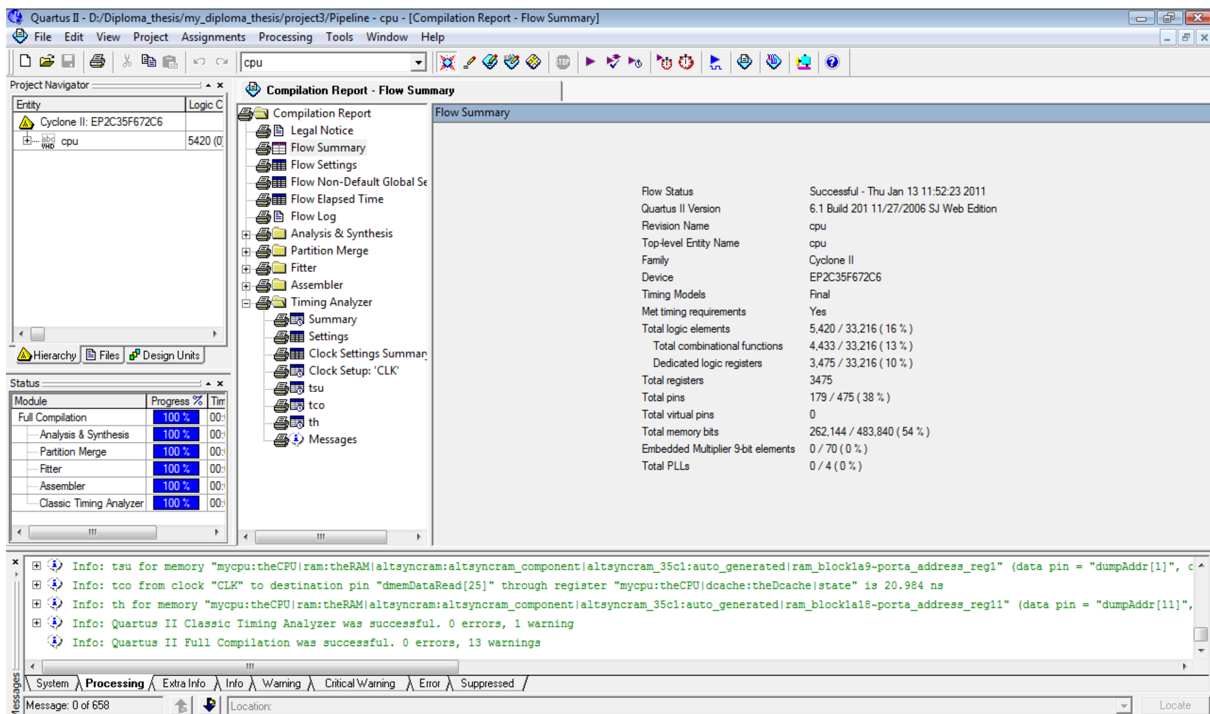
Παρότι η εντολή στην icache είναι έγκυρη δεν έχουμε hit γιατί itagin!=itagStored και δεν είναι αποθηκευμένη στην cache η εντολή που αναζητούμε.

Σε αυτή την περίπτωση έχουμε hit γιατί τα δεδομένα είναι έγκυρα και itagin=itagStored. Άρα η εντολή που αναζητούμε επιστρέφει στο pipeline από την icache.



5.2.2 Σύνθεση και Χρονική Ανάλυση

Έχοντας τελειώσει με τη λογική ανάλυση του pipeline επεξεργαστή προχωρούμε στη σύνθεση προκειμένου να βεβαιωθούμε ότι όντως έχουμε σχεδίαση σωστά το κύκλωμα και έπειτα να το «κατεβάσουμε» στο FPGA ακολουθώντας την ίδια διαδικασία που ακλουθήσαμε για το κύκλωμα του επεξεργαστή single-cycle. Αρχικά δημιουργούμε ένα νέο project ...



Compilation Report - Timing Analyzer Summary

- Compilation Report
  - Legal Notice
  - Flow Summary
  - Flow Settings
  - Flow Non-Default Global Se
  - Flow Elapsed Time
  - Flow Log
  - Analysis & Synthesis
  - Partition Merge
  - Fitter
  - Assembler
  - Timing Analyzer
    - Summary
    - Settings
    - Clock Settings Summar
    - Clock Setup: 'CLK'
    - tsu
    - tco
    - th
    - Messages

Timing Analyzer Summary

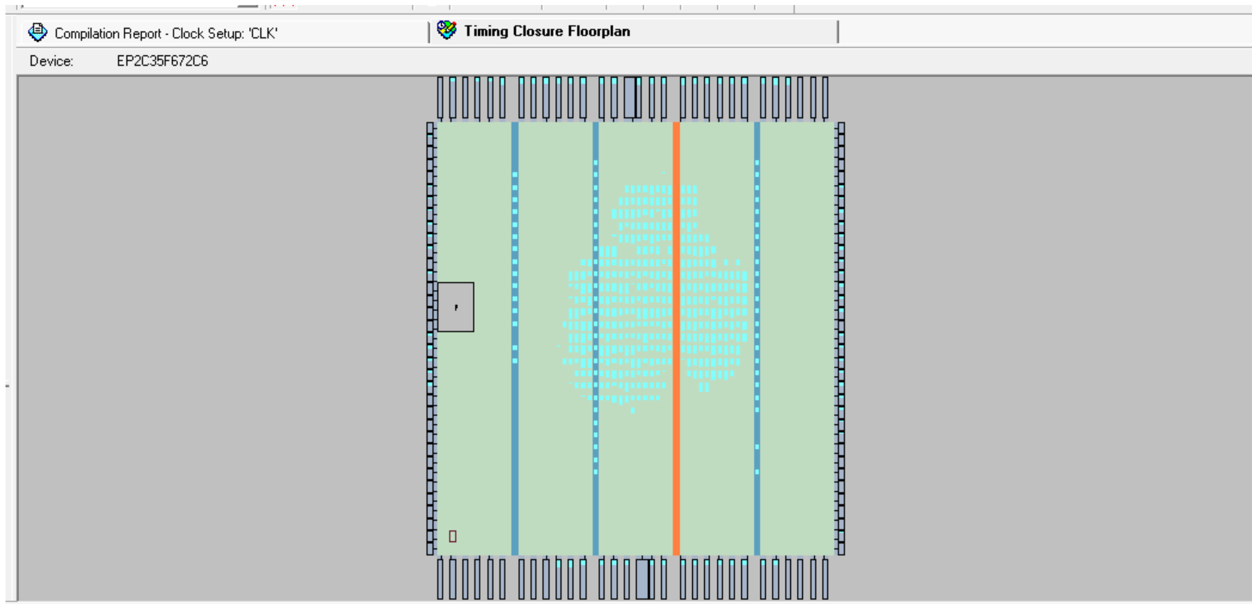
	Type	Slack	Required Time	Actual Time	From	To
1	Worst-case tsu	N/A	None	8.696 ns	dumpAddr[1]	mycpu:theCPU|ram:theRAM|altsyncrar
2	Worst-case tco	N/A	None	20.984 ns	mycpu:theCPU|dcache:theDcache|state	dmemDataRead[25]
3	Worst-case th	N/A	None	-4.067 ns	dumpAddr[11]	mycpu:theCPU|ram:theRAM|altsyncran
4	Clock Setup: 'CLK'	N/A	None	34.62 MHz (period = 28.882 ns)	mycpu:theCPU|registerFile:theRegFile|reg[5][25]	mycpu:theCPU|programCounter:thePC
5	Total number of failed paths					

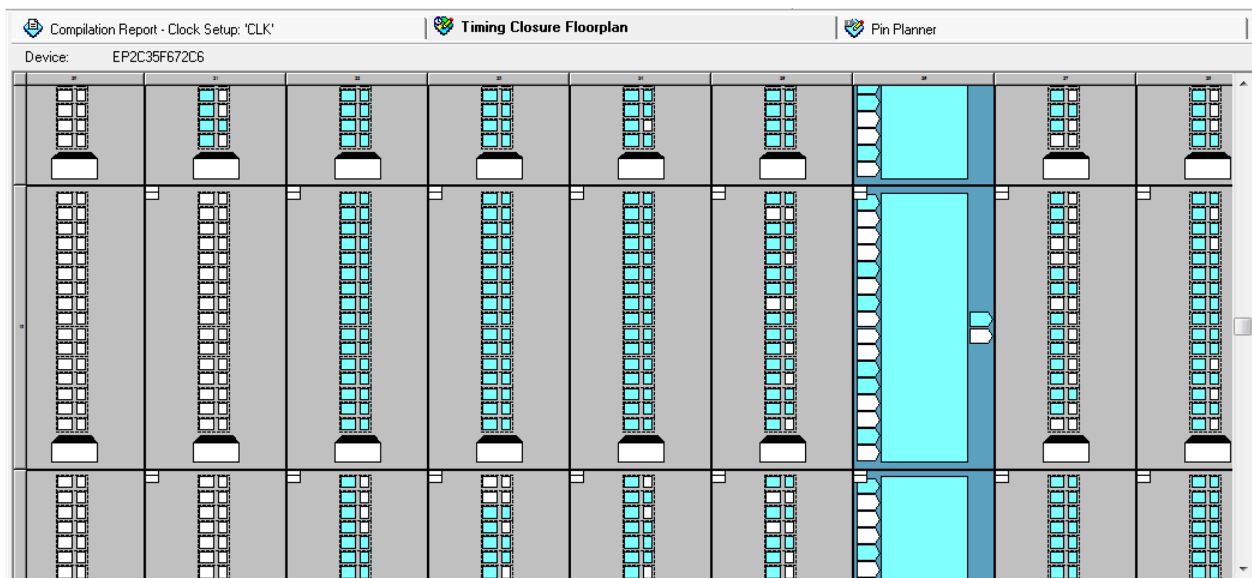
Compilation Report - Clock Setup: 'CLK'

- Compilation Report
  - Legal Notice
  - Flow Summary
  - Flow Settings
  - Flow Non-Default Global Se
  - Flow Elapsed Time
  - Flow Log
  - Analysis & Synthesis
  - Partition Merge
  - Fitter
  - Assembler
  - Timing Analyzer
    - Summary
    - Settings
    - Clock Settings Summar
    - Clock Setup: 'CLK'
    - tsu
    - tco
    - th
    - Messages

Clock Setup: 'CLK'

	Slack	Actual fmax (period)	From	To	From Clock	To Clock
1	N/A	34.62 MHz (period = 28.882 ns)	mycpu:theCPU|registerFile:theRegFile|reg[5][25]	mycpu:theCPU|programCounter:thePC|PC_s[17]	CLK	CLK
2	N/A	34.65 MHz (period = 28.858 ns)	mycpu:theCPU|registerFile:theRegFile|reg[4][15]	mycpu:theCPU|programCounter:thePC|PC_s[17]	CLK	CLK
3	N/A	34.73 MHz (period = 28.796 ns)	mycpu:theCPU|registerFile:theRegFile|reg[23][2]	mycpu:theCPU|programCounter:thePC|PC_s[17]	CLK	CLK
4	N/A	34.79 MHz (period = 28.740 ns)	mycpu:theCPU|registerFile:theRegFile|reg[5][25]	mycpu:theCPU|programCounter:thePC|PC_s[28]	CLK	CLK
5	N/A	34.79 MHz (period = 28.740 ns)	mycpu:theCPU|registerFile:theRegFile|reg[5][25]	mycpu:theCPU|programCounter:thePC|PC_s[29]	CLK	CLK
6	N/A	34.79 MHz (period = 28.740 ns)	mycpu:theCPU|registerFile:theRegFile|reg[5][25]	mycpu:theCPU|programCounter:thePC|PC_s[30]	CLK	CLK
7	N/A	34.79 MHz (period = 28.740 ns)	mycpu:theCPU|registerFile:theRegFile|reg[5][25]	mycpu:theCPU|programCounter:thePC|PC_s[31]	CLK	CLK
8	N/A	34.82 MHz (period = 28.716 ns)	mycpu:theCPU|registerFile:theRegFile|reg[4][15]	mycpu:theCPU|programCounter:thePC|PC_s[28]	CLK	CLK
9	N/A	34.82 MHz (period = 28.716 ns)	mycpu:theCPU|registerFile:theRegFile|reg[4][15]	mycpu:theCPU|programCounter:thePC|PC_s[29]	CLK	CLK
10	N/A	34.82 MHz (period = 28.716 ns)	mycpu:theCPU|registerFile:theRegFile|reg[4][15]	mycpu:theCPU|programCounter:thePC|PC_s[30]	CLK	CLK
11	N/A	34.82 MHz (period = 28.716 ns)	mycpu:theCPU|registerFile:theRegFile|reg[4][15]	mycpu:theCPU|programCounter:thePC|PC_s[31]	CLK	CLK
12	N/A	34.90 MHz (period = 28.654 ns)	mycpu:theCPU|registerFile:theRegFile|reg[23][2]	mycpu:theCPU|programCounter:thePC|PC_s[28]	CLK	CLK
13	N/A	34.90 MHz (period = 28.654 ns)	mycpu:theCPU|registerFile:theRegFile|reg[23][2]	mycpu:theCPU|programCounter:thePC|PC_s[29]	CLK	CLK
14	N/A	34.90 MHz (period = 28.654 ns)	mycpu:theCPU|registerFile:theRegFile|reg[23][2]	mycpu:theCPU|programCounter:thePC|PC_s[30]	CLK	CLK
15	N/A	34.90 MHz (period = 28.654 ns)	mycpu:theCPU|registerFile:theRegFile|reg[23][2]	mycpu:theCPU|programCounter:thePC|PC_s[31]	CLK	CLK
16	N/A	35.12 MHz (period = 28.476 ns)	mycpu:theCPU|registerFile:theRegFile|reg[5][25]	mycpu:theCPU|Ifld_Instruction[0]	CLK	CLK
17	N/A	35.15 MHz (period = 28.452 ns)	mycpu:theCPU|registerFile:theRegFile|reg[4][15]	mycpu:theCPU|Ifld_Instruction[0]	CLK	CLK
18	N/A	35.22 MHz (period = 28.390 ns)	mycpu:theCPU|registerFile:theRegFile|reg[23][2]	mycpu:theCPU|Ifld_Instruction[0]	CLK	CLK
19	N/A	35.32 MHz (period = 28.316 ns)	mycpu:theCPU|registerFile:theRegFile|reg[5][25]	mycpu:theCPU|Ifld_Instruction[23]	CLK	CLK
20	N/A	35.32 MHz (period = 28.316 ns)	mycpu:theCPU|registerFile:theRegFile|reg[5][25]	mycpu:theCPU|Ifld_Instruction[25]	CLK	CLK
21	N/A	35.32 MHz (period = 28.316 ns)	mycpu:theCPU|registerFile:theRegFile|reg[5][25]	mycpu:theCPU|Ifld_Instruction[21]	CLK	CLK
22	N/A	35.32 MHz (period = 28.316 ns)	mycpu:theCPU|registerFile:theRegFile|reg[5][25]	mycpu:theCPU|Ifld_Instruction[22]	CLK	CLK





5.3 Συμπεράσματα

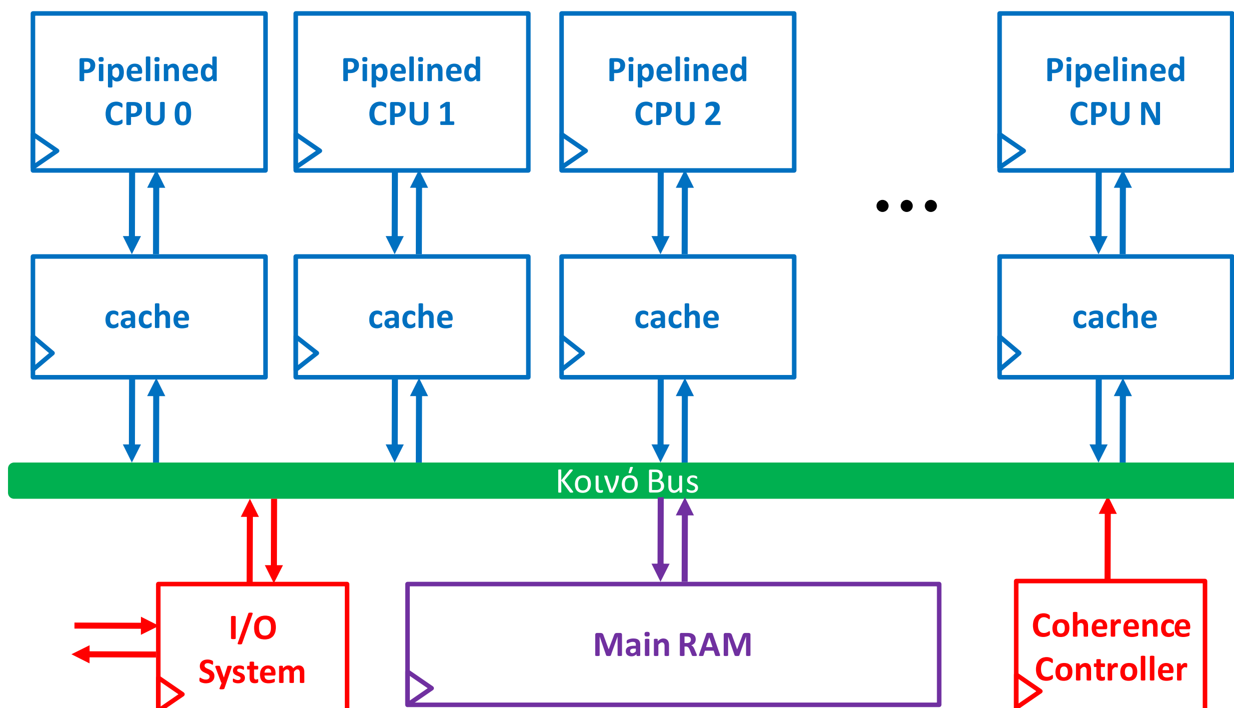
Η υλοποίηση του επεξεργαστή pipeline είναι κρίσιμη για την υλοποίηση του επεξεργαστή multicore. Για τον λόγο αυτό αφιερώσαμε αρκετό χρόνο έτσι ώστε να είμαστε σίγουροι για την ορθή λειτουργικότητά του. Έτσι γράψαμε assembly κώδικα για πολλούς αλγορίθμους όπως quick-sort, Fibonacci, seach και multiplication οι οποίοι έδιδαν σωστά τα αποτελέσματα στη μνήμη δεδομένων. Όσον αφορά την απόδοση αυτού του επεξεργαστή βλέπουμε ότι είναι αρκετά πιο γρήγορος από τον single-cycle όπως αναμενόταν.

6. Multi-core processor

Οι επεξεργαστές με πολλά cores έχουν ένα κοινό bus από όπου μπορούν να διαβάζουν και να γράφουν δεδομένα στην κύρια μνήμη RAM. Μέσω αυτού του bus γίνεται και η μεταξύ τους επικοινωνία. Έτσι αν κάποιο core διαβάσει κάποια δεδομένα από τη RAM (μέσω lw εντολών) και τα τροποποιήσει τότε όλα τα άλλα cores θα πρέπει να είναι ενήμερα για αυτή την πράξη. Τα τροποποιημένα δεδομένα βρίσκονται στην cache του εκάστοτε core και όχι στην κύρια μνήμη. Αν βρίσκονταν στην κύρια μνήμη τότε δεν θα υπήρχε κανένα πρόβλημα γιατί όλα τα cores θα διάβαζαν τα σωστά δεδομένα. Βέβαια ο λόγος ύπαρξης των δεδομένων στην cache όπως έχουμε αναφέρει στο Κεφ. 5 είναι η αύξηση της ταχύτητας του επεξεργαστή.

Το παραπάνω πρόβλημα αναφέρεται ως cache coherence problem. Έτσι, δεδομένου της παραπάνω κατάστασης η λύση του προβλήματος δίδεται με τη χρήση ενός coherence controller ο οποίος διαβάζει το bus και ανάλογα με τις εντολές που έχουν ζητηθεί από τα cores δίνει τις κατάλληλες προτεραιότητες.

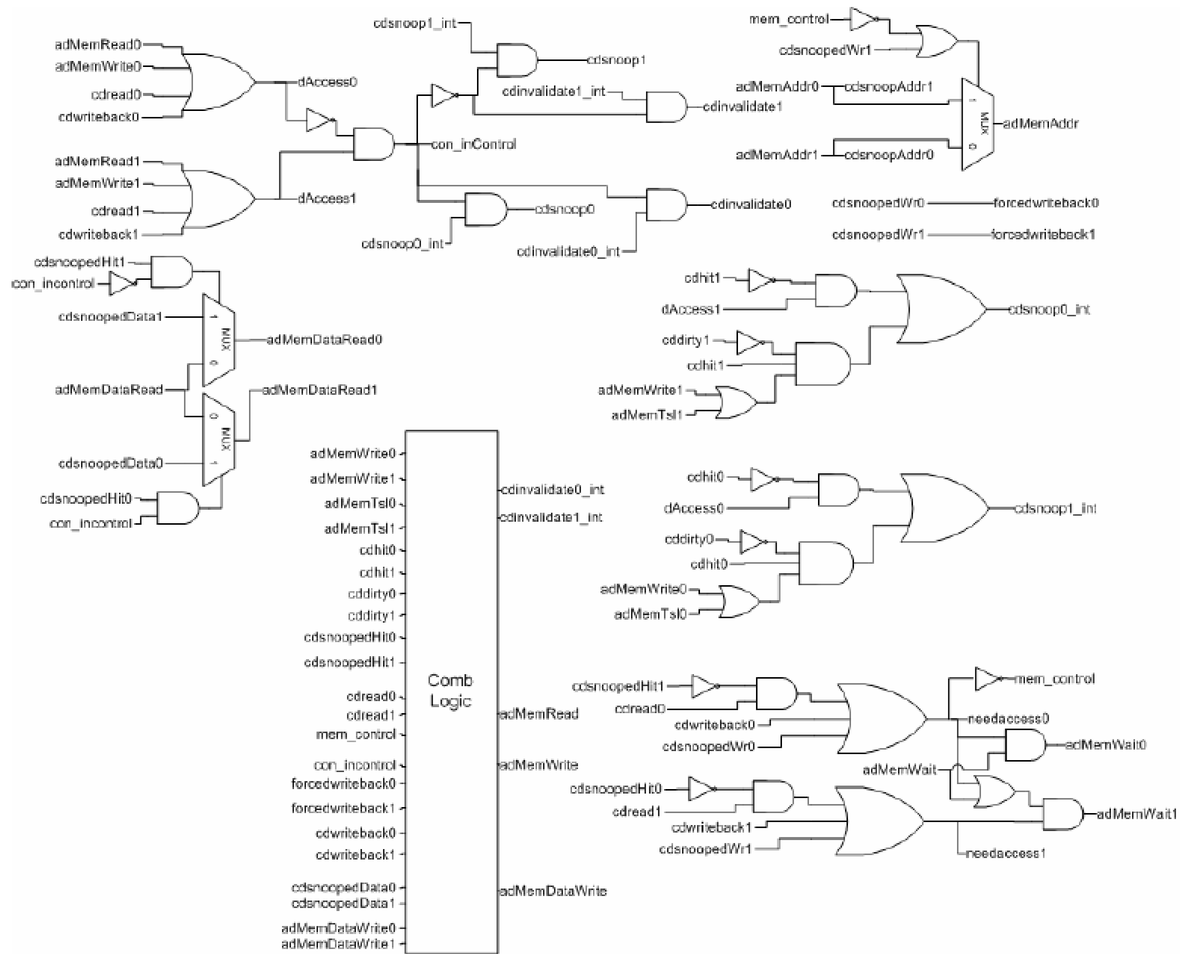
Στο Σχ. 6-1.1 βλέπουμε το σχηματικό διάγραμμα ενός επεξεργαστή πολλών cores και πως αυτοί επικοινωνούν μέσω του κοινού bus για να διαβάζουν και να γράφουν δεδομένα από την κύρια μνήμη. Στο ίδιο σχήμα βλέπουμε τον coherence controller ο οποίος είναι υπεύθυνος για την ορθότητα των δεδομένων. Επίσης, το bus επικοινωνεί και με συσκευές εισόδου/εξόδου (π.χ. printf, scanf). Σε αυτή τη διπλωματική εργασία έχουμε υλοποιήσει τον coherence controller αλλά όχι το σύστημα εισόδου/εξόδου.



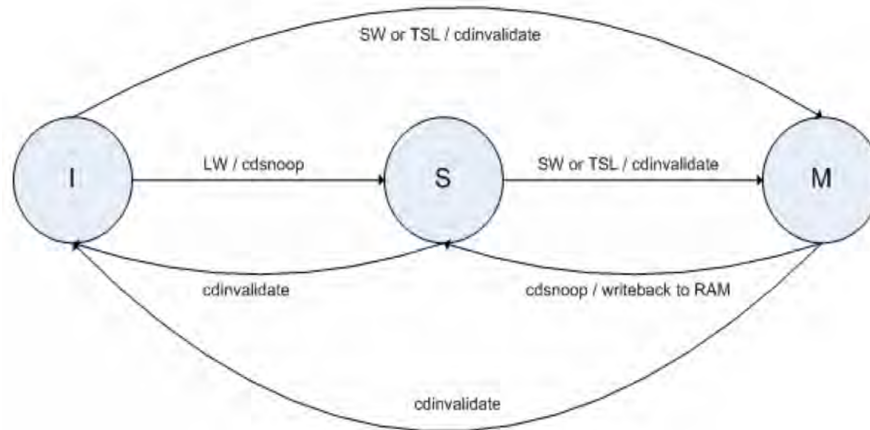
Σχ. 65.31-1: Σχηματικό διάγραμμα ενός επεξεργαστή με N cores οι οποίοι επικοινωνούν μεταξύ τους μέσω του κοινού bus. Η ορθότητα των δεδομένων επιβάλλεται από τον coherence controller.

6.1 Coherence Controller

Ο coherence controller όπως έχουμε προαναφέρει είναι το κύκλωμα που είναι υπεύθυνο για τη συνεκτικότητα των δεδομένωνπου είναι αλλαγμένα στις dcaches των cores. Το σχηματικό διάγραμμα του κυκλώματος φαίνεται στην παρακάτω εικόνα.

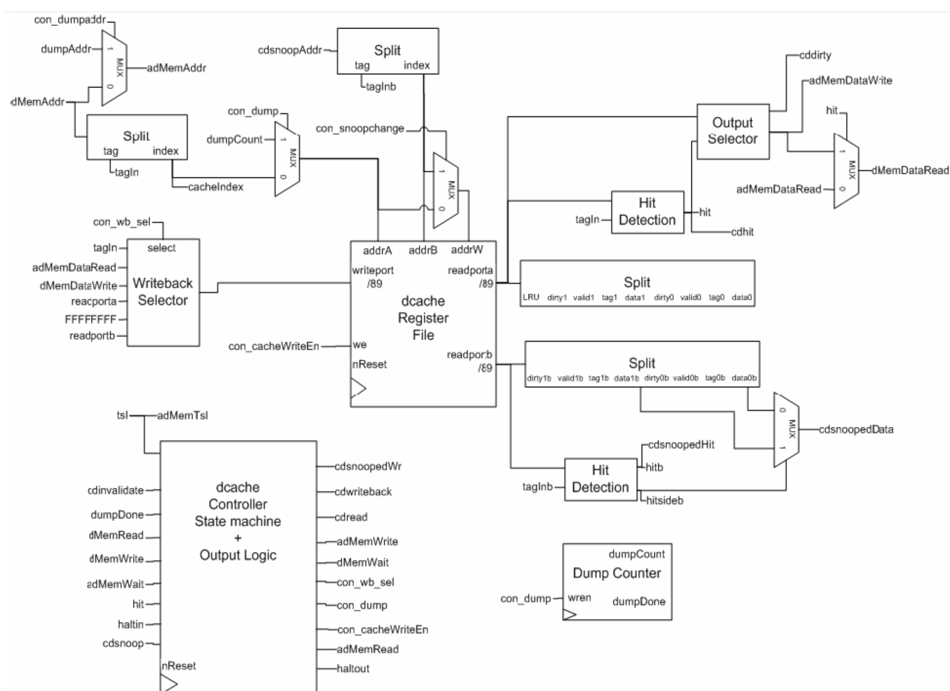


Ο coherence controller υλοποιεί το προτόκολλο MSI που είναι μια μηχανή καταστάσεων. Η αναπαράστηαση της μηχανής αυτής παρουσιάζεται στο παρακάτω σχήμα. Το MSI έχει τρεις καταστάσεις την invalid (I), την share (S) και την modify (M).



Σχ. 6-1.2:MSI πρωτόκολο.

6.2 Μνήμη δεδομένων για dual-core επεξεργαστή

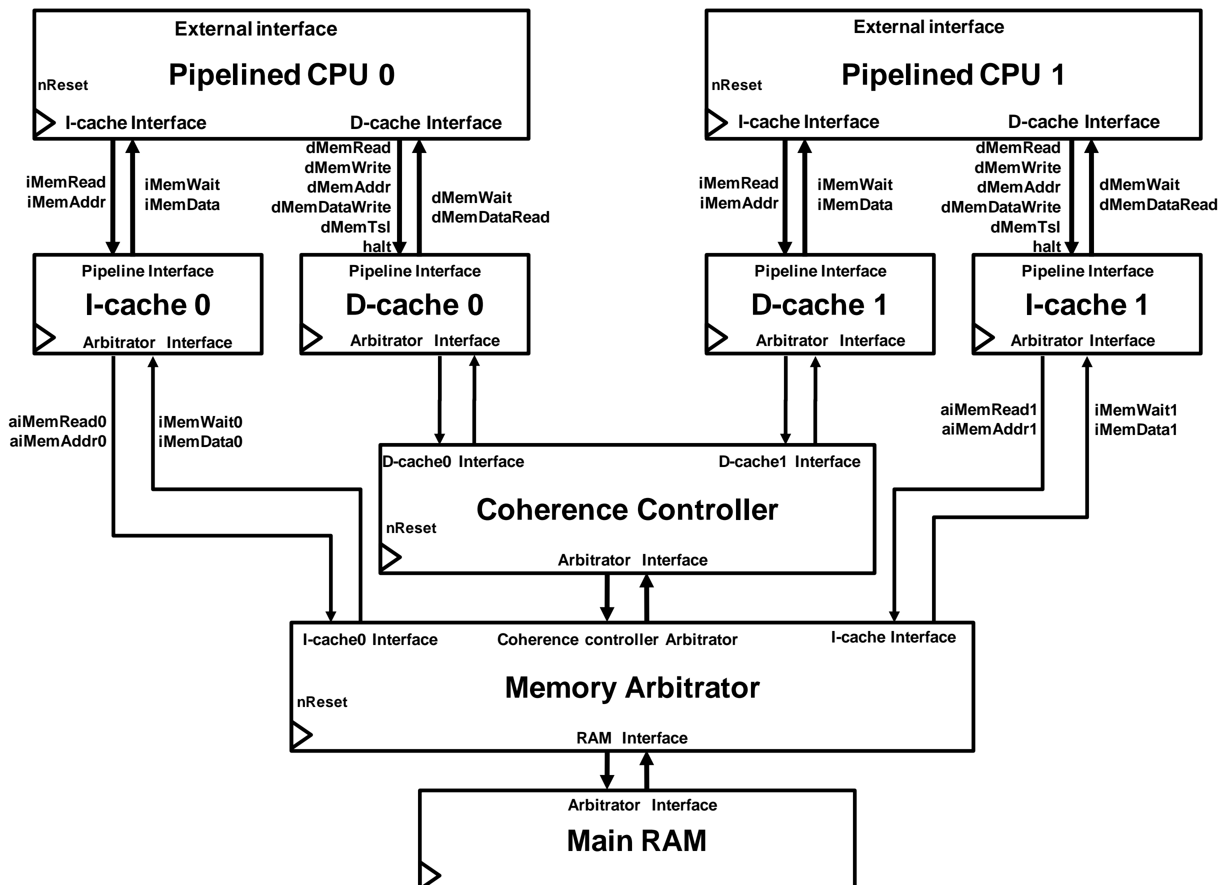


6.3 Σχηματικό Διάγραμμα του Dual-Core

Όπως έχουμε προαναφέρει στην εισαγωγή ο multi-core επεξεργαστής που υλοποιήσαμε σε αυτή την εργασία έχει δύο cores. Η αρχιτεκτονική του κάθε core είναι η ίδια με αυτή του επεξεργαστή pipeline που παρουσιάσαμε στο Κεφ. 5. Η μόνη τροποποίηση που έχουμε κάνει (βλ. προηγούμενη

ενότητα) είναι στη μνήμη δεδομένων (dcache) η οποία έχει πλέον δύο πόρτες διαβάσματος ενώ η αντίστοιχη μνήμη του επεξεργαστή pipeline είχε μόνο μία.

Στο Σχ. 6.2-1 έχουμε σχεδιάσει με λεπτομέρεια το σχηματικό διάγραμμα των δύο cores καθώς και τον τρόπο με τον οποίο γίνεται η επικοινωνία. Παρατηρούμε ότι μόνο οι μνήμες δεδομένων επικοινωνούν με τον coherence controller. Ο λόγος που επιλέξαμε αυτή τη σχεδίαση είναι γιατί μόνο οι dcache έχουν τη δυνατότητα να τροποποιήσουν τα δεδομένα τους για τα οποία ο coherence controller είναι υπεύθυνος για τη συνοχή τους. Οι icache και ο coherence controller επικοινωνούν με τον arbitrator ο οποίος δίνει κατάλληλες προτεραιότητες μεταξύ των μνημών δεδομένων και εντολών. Προφανώς ο arbitrator επικοινωνεί άμεσα με την κύρια μνήμη.

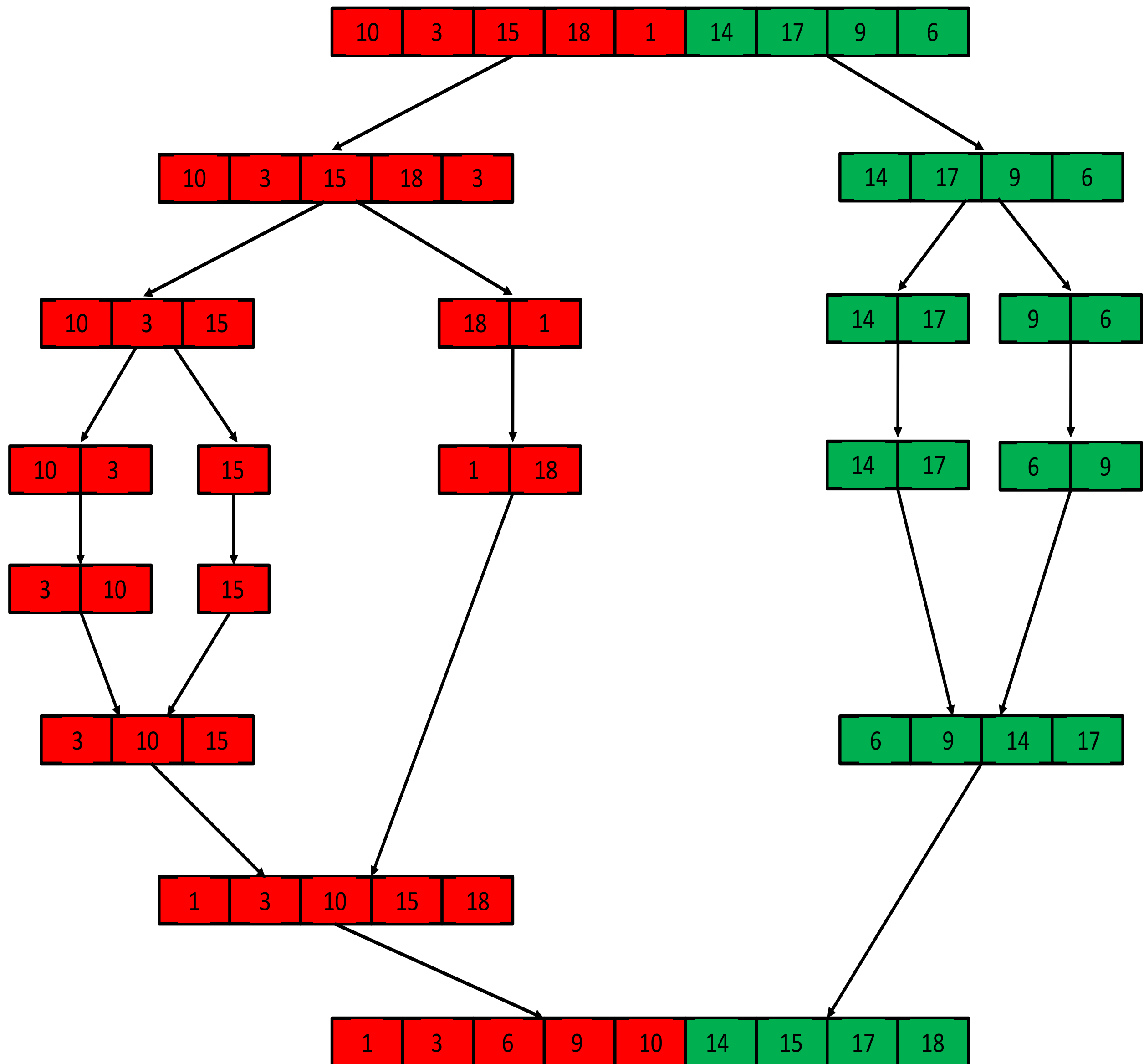


Σχ. 6.2-1: Σχηματικό διάγραμμα του επεξεργαστή dual-core. Τα σήματα για την επικοινωνία τόσο των icache όσο και των dcache με τον coherence controller και τον memory arbitrator έχουν σχεδιαστεί με λεπτομέρεια.

6.4 Το παράλληλο πρόγραμμα mergesort

Η ορθή λειτουργία του επεξεργαστή dual-core αλλά και ο τρόπος λειτουργίας του γίνεται με την εκτέλεση του προγράμματος mergesort. Παρότι εκτελέσαμε μια πληθώρα από πολύ απλά μέχρι πολύ πολύπλοκα προγράμματα για να βεβαιωθούμε για την ορθότητα της λειτουργίας του επεξεργαστή θα παρουσιάσουμε το mergesort γιατί είναι ένα αντιπροσωπευτικό παράδειγμα ενός ταυτόχρονου προγράμματος. Το συγκεκριμένο πρόγραμμα mergesort διαβάζει 100 μη ταξινομημένους αριθμούς που είναι αποθηκευμένοι στη μνήμη RAM και μετά το πέρας της εκτέλεσής του επιστρέφει, δηλαδή

γράφει ξανά στη μνήμη RAM τους 100 αριθμούς ταξινομημένους σε αύξουσα σειρά. Ο λόγος που χρησιμοποιούμε τον αλγόριθμο mergesort είναι γιατί μπορεί να εκτελεστεί ταυτόχρονα και στους δύο cores του επεξεργαστή. Στο Σχ. 6.4-1 δίνουμε ένα παράδειγμα με εννέα αριθμούς και πώς ο αλγόριθμος διχοτομεί τον πίνακα ώστε τελικά να γίνει η ταξινόμησή του. Με κόκκινο χρώμα είναι τα βήματα που εκτελούνται στο core0 του επεξεργαστή και με πράσινο τα βήματα που εκτελούνται στο core1. Είναι σημαντικό να προσέξουμε ότι το τελευταίο merge γίνεται στο core0. Αυτό σημαίνει ότι ο core0 εφόσον τελειώσει πρώτα από τον core1 θα πρέπει να τον περιμένει. Αν αυτό δεν συμβεί, τότε η τελική ταξινόμηση θα είναι λάθος. Πρόκειται εδώ για ένα κλασσικό πρόβλημα συγχρονισμού.



Σχ. 6.4-1Παρουσίαση των βημάτων του αλγορίθμου mergesort για την ταξινόμηση ενός πίνακα αριθμών εννέα θέσεων.

Παρακάτω θα παραθέσουμε τμήματα της assembly που γράψαμε για την εκτέλεση του παραπάνω αλγορίθμου. Είναι προφανές ότι ο κάθε core του επεξεργαστή θα έχει το δικό κώδικα και κατά

συνέπεια οι μετρητές εντολών θα αρχικοποιούνται σε διαφορετικές τιμές. Αυτό βέβαια όπως θα δούμε παρακάτω καθορίζεται από εμάς στο αρχείο της assembly. Ο κώδικας για τον πρώτο core είναι:

```
# Code for the Core0 Processor
org      0x0000            # Program counter initialization for core0
ori      $sp, $0, 0x6FFC # stack
ori      $s0, $0, 0x64   # length of data array = 100
srl      $s1, $s0, 1     # length of array to sort on core0 (100/2)
or      $s2, $0, $0     # pointer to first element to sort
addiu   $s3, $s1, -1    # pointer to last element to sort
jal     mergesort    # call mergesort to sort the first half array

# core0 is done and waits for Core1 to finish. The address exchange
# is the common place where the cores exchange messages.
ori      $t1, $0, exchange
core0wait:
# Load the message from core1
lw $t0, 0($t1)
# If there is no message this means that core1 still working then
# it should wait.
beq      $t0, $0, core0wait

# At this point both cores are done. The last merge will performed.
# take into account the flag state of each core
# Flag == 1 --> sorted array is in inputdata
# Flag == 0 --> sorted array is in augdata

ori      $gp, $0, retdata  # final output is retdata

bne     $a3, $t3, c0flagl1      # if Flag == 1
ori      $k0, $0, inputdata
j       c0flagl2
c0flagl1:                               # else
ori      $k0, $0, augdata
c0flagl2:
# load the flag state of Core 1
lw $a3, 4($t1)

bne     $a3, $t3, c0flagu1      # if Flag == 1
ori      $k1, $0, inputdata
j       c0flagu2
c0flagu1:                               # else
ori      $k1, $0, augdata
c0flagu2:

# do the final merge using only core 0
ori      $a0, $0, 0        # low = 0
ori      $a1, $s3, 0 # mid = len/2-1
addiu    $a2, $s0, -1      # high = len-1
```

```
    # perform the last merge
    jal      merge       # Call the final merge.
    halt #Terminate core0.Since both cores are halted the procesr halts.
```

Προτού δείξουμε την assembly για τις συναρτήσεις merge και mergesort ας δούμε τον κώδικα του δεύτερου core1. Είναι ενδιαφέρον να δούμε πως γίνεται η επικοινωνία ώστε να επιτύχουν τον κοινό τους σκοπό.

```
# Code for the Core1 Processor
org 0x200                    # Program counter initialization for core1
ori     $sp, $0, 0x7FFC      # stack pointer
ori     $s0, $0, 0x64        # length of data array = 100
srl     $s2, $s0, 1          # first element to sort
subu    $s1, $s0, $s2        # length of array to sort on core1
addiu   $s3, $s0, -1         # last element to sort

jal     mergesort

# set flag value to RAM for passing message to core0
ori   $t0, $0, exchange
sw    $a3, 4($t0)
# set done value in RAM so that core 0 can go on with the final merge.
sw    $t3, 0($t0)
halt                         # Terminate core1.
```

Παρακάτω παραθέτουμε τον κώδικα για τη συνάρτηση mergesort που χρησιμοποιείται και από τους δύο cores και για τη συνάρτηση merge που χρησιμοποιείται μόνο από τον core0. Να σημειωθεί ότι παρότι η mergesort χρησιμοποιείται και από τα δύο cores υπάρχει στη μνήμη RAM μόνο μια φορά αλλά φορτώνεται και στις caches εντολών και των δύο cores. Δεν θα εξηγήσουμε τις λεπτομέρειες αυτού του κώδικα γιατί θεωρείται ότι είναι γνωστή η υλοποίησή του από το μάθημα «Εισαγωγή στους επεξεργαστές» και δεν είναι σκοπός αυτής της διπλωματικής εργασίας. Ο κώδικας για τη συνάρτηση mergesort είναι:

```
mergesort:
ori     $30, $ra, 0     # copy return address
ori     $a3, $0, 1      # Flag
ori     $s4, $0, 1      # N_size = merge size for this run
ori     $t3, $0, 1      # constant 1

msortloop1:
slt     $t0, $s4, $s1    # while (N_size < Size)
bne     $t0, $t3, msortend1
# set input and ouput addresses based on Flag
bne   $a3, $t3, msrev1      # if Flag == 1, inputdata --> augdata
ori   $k0, $0, inputdata
ori   $k1, $0, inputdata
ori   $gp, $0, augdata
```

```
j       msrev2

msrev1:                  # else augdata --> inputdata
ori   $k0, $0, augdata
ori   $k1, $0, augdata
ori   $gp, $0, inputdata

msrev2:
subu    $s7, $s1, $s4   # s7 = Size - N_Size
addu    $s7, $s7, $s2   # s7 = Size - N_Size + first
sll     $s6, $s4, 1     # N_Size_next = N_Size * 2
ori     $a0, $s2, 0     # I, mergesort loop counter, I = first

msortloop2:
slt     $t0, $a0, $s7   # while (I < Size-N_Size)
bne     $t0, $t3, msortend2
addu    $a2, $s6, $a0
addiu   $a2, $a2, -1     # high = I + 2*N_Size - 1
slt     $t0, $s3, $a2   # if (last < high)
bne     $t0, $t3, msortnext1
ori     $a2, $s3, 0     # high = last

msortnext1:
addu    $a1, $a0, $s4   # mid = I + N_Size - 1
addiu   $a1, $a1, -1
jal     merge           # merge the two sides
addu    $a0, $a0, $s6   # I += 2 * N_Size
j       msortloop2

msortend2:
# copy the end of the input array that was missed
bne     $a3, $t3, mcopy1      # if Flag == 1, inputdata --> augdata

mcstart1:
slt     $t0, $a2, $s3        # if (high < last)
bne     $t0, $t3, mcopy2
addiu $a2, $a2, 1          # high++
sll     $t2, $a2, 2          # t2 = high * 4 (byte offset)
lw      $t1, inputdata($t2)  # augdata[high] = inputdata[high]
sw      $t1, augdata($t2)
j       mcstart1
mcopy1:                              # else augdata --> inputdata
slt     $t0, $a2, $s3         # if (high < last)
bne     $t0, $t3, mcopy2
addiu $a2, $a2, 1          # high++
sll     $t2, $a2, 2          # t2 = high * 4 (byte offset)
lw      $t1, augdata($t2)    # inputdata[high] = augdata[high]
sw      $t1, inputdata($t2)
j       mcopy1
```

```
mcopy2:
#     ori     $k0, $0, 2
#     bne     $s4, $k0, mnexttest
#     halt

mnexttest:
ori     $s4, $s6, 0     # N_Size *= 2
xori    $a3, $a3, 1     # Flag ^= 1
j       msortloop1

msortend1:
ori     $ra, $30, 0     # copy return address back
jr      $ra
```

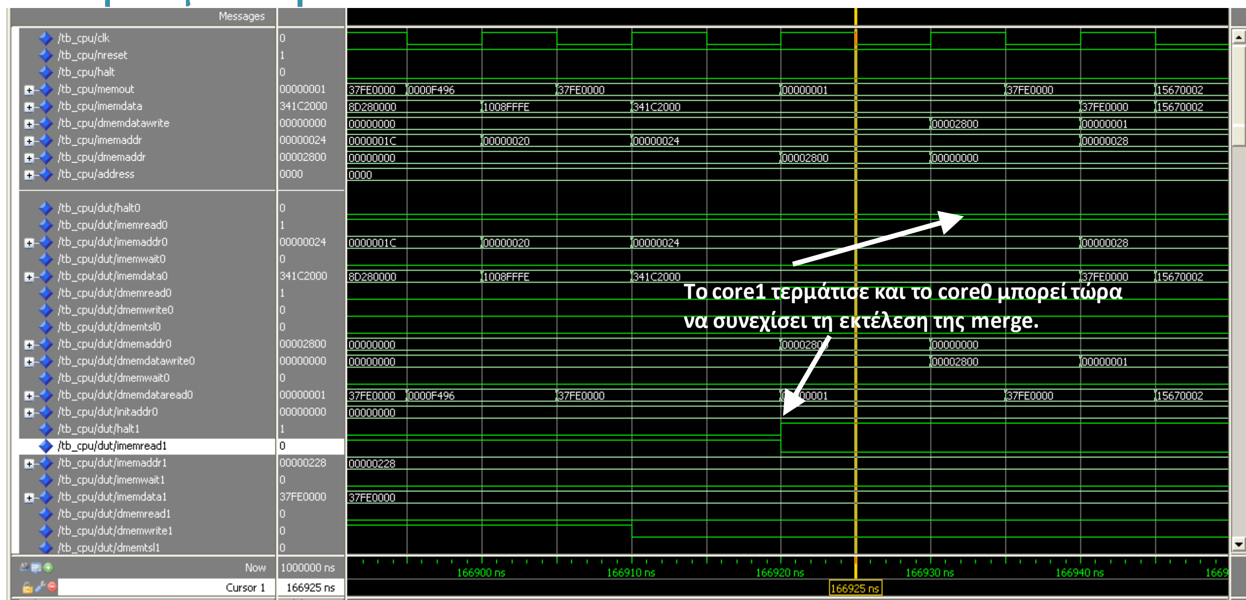
Ο κώδικας για τη συνάρτηση merge ο οποίος χρησιμοποιείται και από τη mergesort για τα ενδιάμεσα βήματα merge δίνεται παρακάτω:

```
merge:
#  $a0 - Low,  $a1 - Mid,  $a2 - High, $a3 - Flag
#  $k0 - lower input array address, $k1 - upper input array address
#  $gp - output array address
or      $t0, $a0, $0    # i, upper half index
addiu   $t1, $a1, 1     # j, lower half index
or      $t2, $a0, $0    # k, loop counter
ori     $t3, $0, 1      # constant 1

mergeloop:
slt     $t6, $a2, $t2   # high < k
beq     $t6, $t3, mergeend # exit if not k <= high
sll     $t4, $t0, 2             # t4 = 4 * i
sll     $t5, $t1, 2             # t5 = 4 * j
addu  $t4, $t4, $k0           # t4 = lower input address + 4i
addu  $t5, $t5, $k1           # t4 = upper input address + 4i
lw      $t4, 0($t4)           # in[i]
lw      $t5, 0($t5)           # in[j]


slt     $t6, $a2, $t1   # if (j > high)
bne     $t6, $t3, merge2
ori     $t7, $t4, 0           # out[k] = in[i]
addiu   $t0, $t0, 1     # i++
j       mergedone

merge2:
slt         $t6, $a1, $t0     # if (i > mid)
bne         $t6, $t3, merge3
ori         $t7, $t5, 0       # out[k] = in[j]
addiu       $t1, $t1, 1       # j++
j           mergedone
```

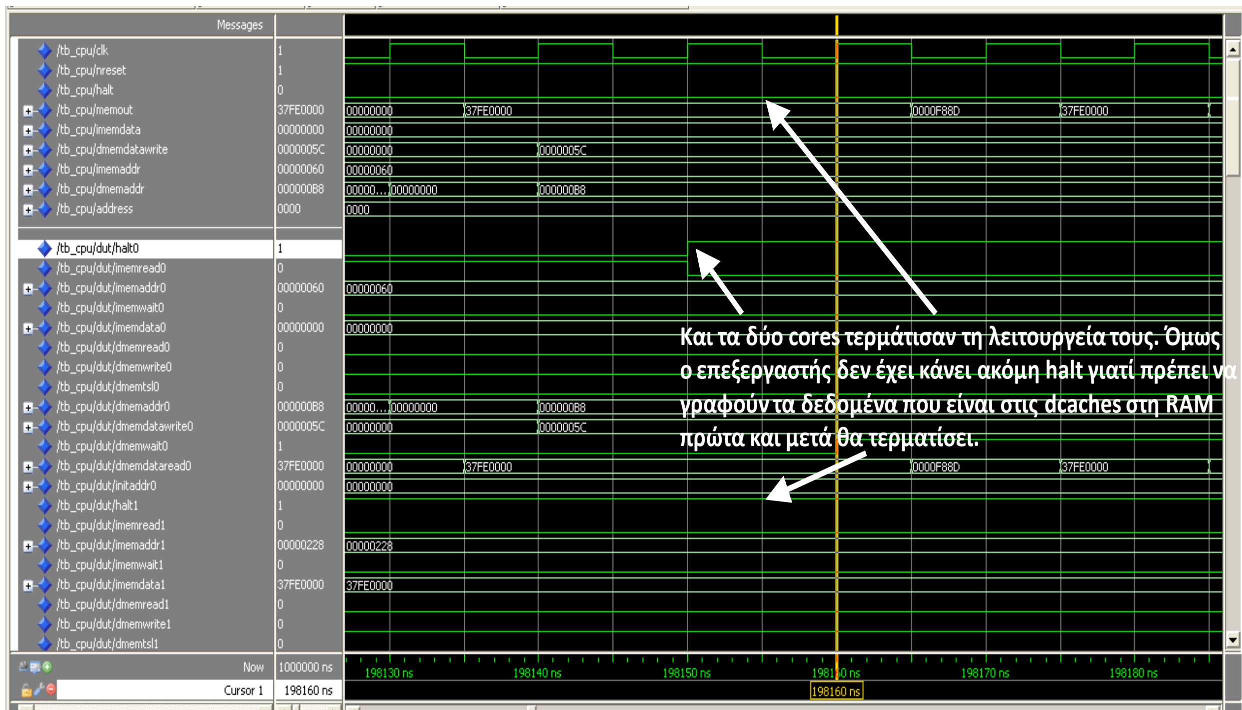
```
merge3:
slt         $t6, $t5, $t4          # if (in[i] > in[j])
bne         $t6, $t3, merge4
ori         $t7, $t5, 0       # out[k] = in[j]
addiu       $t1, $t1, 1       #j++
j           mergedone

merge4:
ori         $t7, $t4, 0       # out[k] = in[i]
addiu       $t0, $t0, 1       # i++

mergedone:
sll         $t4, $t2, 2       # t4 = 4 * k
addu        $t4, $t4, $gp          # t4 = output address + 4k
sw          $t7, 0($t4)
addiu       $t2, $t2, 1       # k++
j           mergeloop

mergeend:
jr          $ra               #end of merge function, return to mergesort.
```

Σε αυτό το σημείο θα θέλαμε να τονίσουμε πόσο σημαντικός ήταν ο τρόπος με τον οποίο ελέγχουμε την ορθή λειτουργία των επεξεργαστών όπου ελέγχουμε αν τα αποτελέσματα που έχουν γραφεί στη μνήμη είναι τα σωστά. Επίσης η αρχικοποίηση της μνήμης σύμφωνα με το αρχείο meminit.hex απλοποιεί σημαντικά τον έλεγχο της ορθότητας του επεξεργαστή, γιατί διαφορετικά θα έπρεπε να τροποποιούμε όλο το testbench κάθε φορά που κάνουμε μια αλλαγή στον κώδικα της assembly.

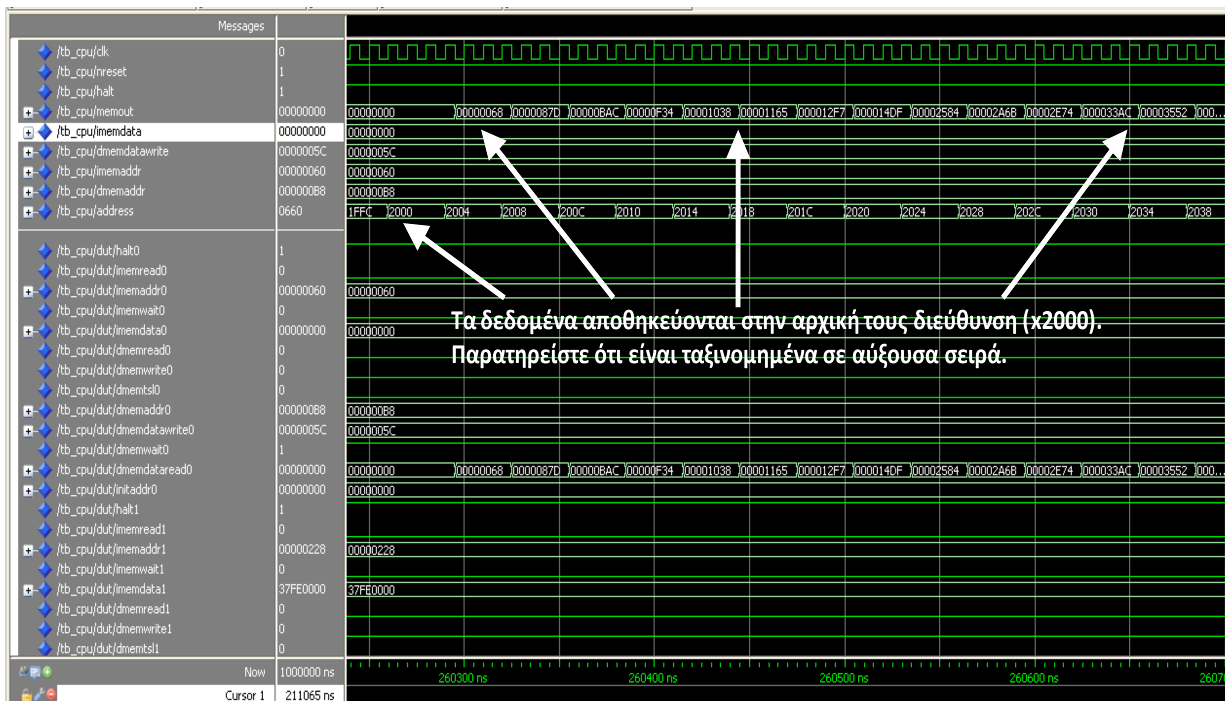
6.5 Προσομοίωση



Εικ. 6.5-1: Συγχρονισμός των core για την ορθή λειτουργία της mergesort.



Εικ. 6.5-2: Τερματισμός και των δύο cores αλλά ο επεξεργαστής multicore συνεχίζει να τρέχει μέχρι να γράψει όλα τα δεδομένα που βρίσκονται στις caches στην RAM ώστε το testbench να γράψει το σωστό αρχείο.



Εικ. 6.5-3: Ο επεξεργαστής τερμάτισε τη λειτουργία του και έτσι το testbench γράφει το αρχείο εξόδου. Παρατηρήστε ότι τα δεδομένα είναι ταξινομημένα.

6.6 Συμπεράσματα

Ο επεξεργαστής dual-core ήταν ο πιο δύσκολος τόσο στην υλοποίησή του όσο και στην επαλήθευσή του από τους άλλους δυο επεξεργαστές. Παρότι χρησιμοποιήσαμε για κάθε core τους επεξεργαστές pipeline που παρουσιάστηκε στο Κεφ. 5, η επικοινωνία των μνημών και το coherency των δεδομένων ήταν μια επίπονη διαδικασία. Τελικά, και ο επεξεργαστής dual-core που ήταν ο τελικός σκοπός αυτής της εργασίας υλοποιήθηκε με επιτυχία τόσο σε λογικό επίπεδο όσο και σε πραγματική υλοποίηση στο FPGA.

7. FPGA

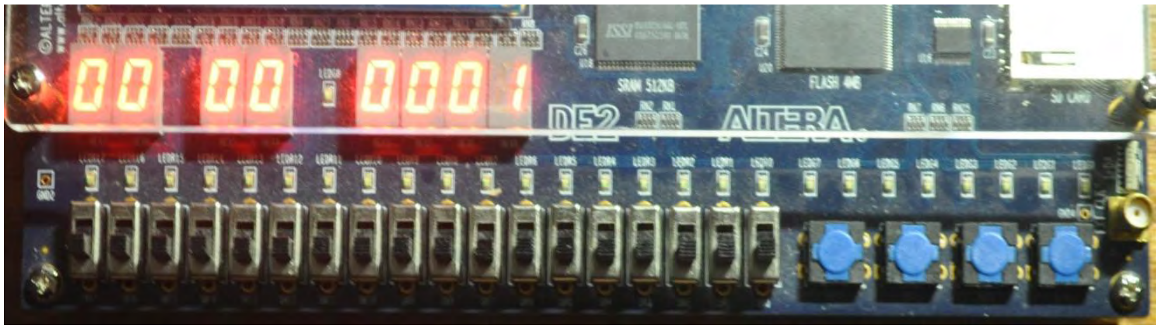
Κατά τη διάρκεια της σύνθεσης του κυκλώματος δημιουργήθηκε από τον Assembler του Quartus ένα αρχείο .sof. Το αρχείο αυτό περιέχει τη σχεδίαση του επεξεργαστή και φορτώνεται στο FPGA που φαίνεται στην Εικ. 7-1.

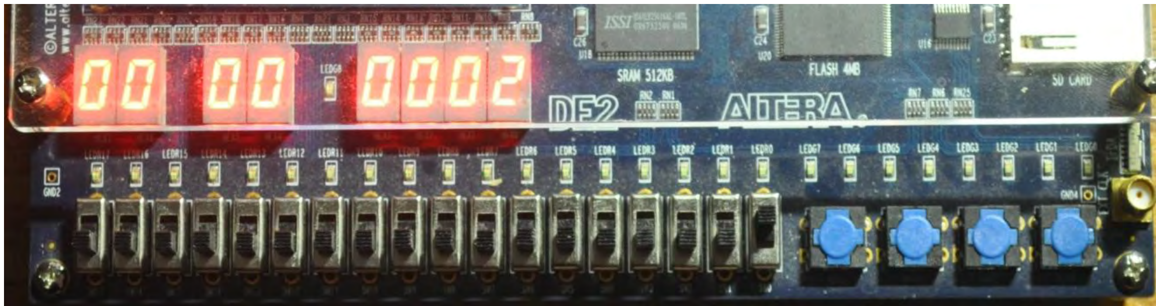
Εικ. 7-1: Το Altera DE2 borad που χρησιμοποιήσαμε για να κατεβάσουμε τις τρεις σχεδιάσεις των επεξεργαστών που παρουσιάστηκαν στα Κεφ. 4, 5 και 6.

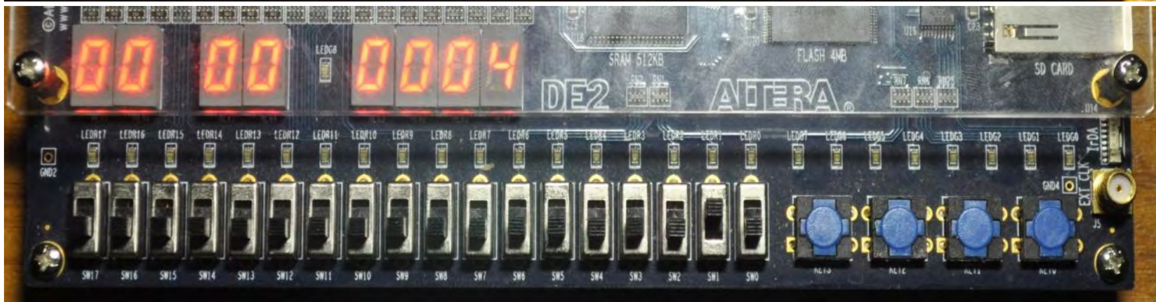
Αφήνοντας το κουμπί reset ο επεξεργαστής εκτελεί το πρόγραμμα που του φορτώθηκε στη μνήμη του και γράφει τα αποτελέσματα στη μνήμη δεδομένων. Το Quartus μας δίνει τη δυνατότητα να ελέγξουμε τα δεδομένα που είναι γραμμένα στη μνήμη. Έτσι στην Εικ. 7-1: βλέπουμε ότι ο επεξεργαστής εκτέλεσε ορθά το πρόγραμμα του πολλαπλασιασμού που παρουσιάσαμε στο Κεφ. 3.

Παρακάτω θα παρουσιάσουμε μια σειρά από σχεδιάσεις που αποδεικνύουν ότι τα κυκλώματα που σχεδιάσαμε λειτουργούν σωστά όχι μόνο σε επίπεδο προσομοίωσης αλλά και στην πραγματική τους υλοποίηση σε FPGA. Αρχικά θα παρουσιάσουμε τη ALU. Για την ALU τα KEY0, KEY1 και KEY2 αντιστοιχούν στο opcode της ALU. Το KEY3 ενεργοποιεί την εγραφή της εισόδου A σε έναν καταχωρητή και η είσοδος B δίνεται από τα switches0-17 για τα 18 λιγότερο σημαντικά bits του A. Τα άλλα τα θεωρούμε μηδέν.

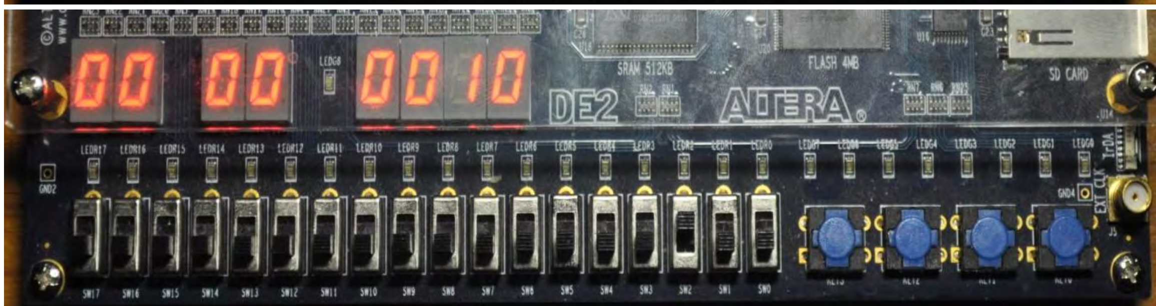
Έτσι αρχικά αποθηκεύουμε τον αριθμό 1 στην είσοδο A και δίνουμε στο B αριθμούς απο 1 μέχρι πέντε και επειδή το opcode της ALU είναι b000 θα εκτελείται η πράξη SLL. Αυτή η διαδικασία παρουσιάζεται στις παρακάτω εικόνες.

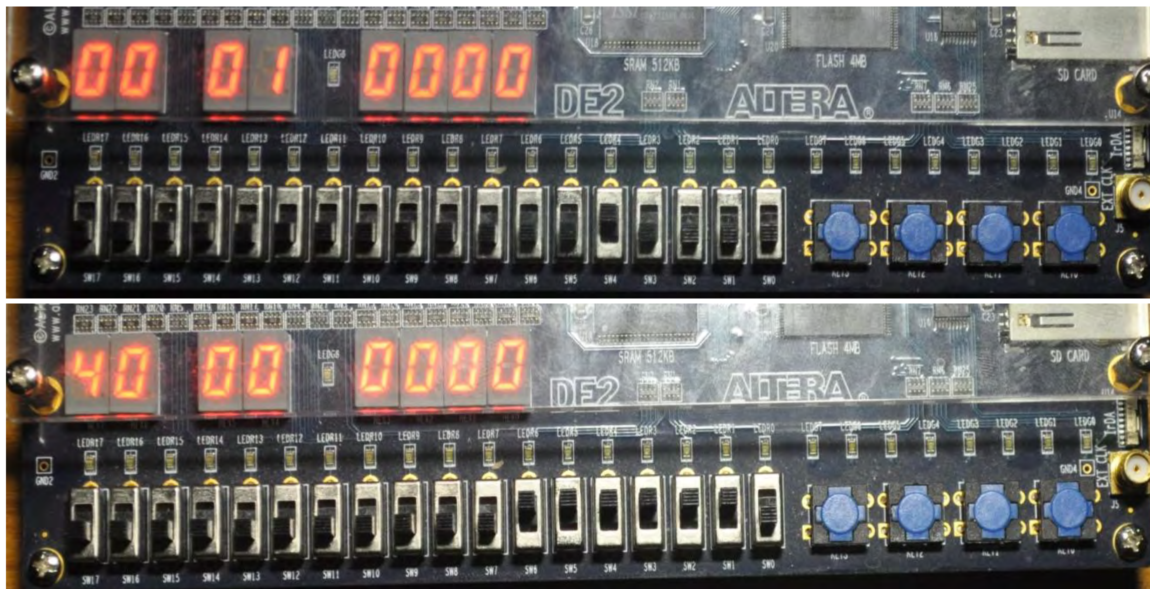

00 00 0001
SRAM 512KB
FLASH 4MB
DE2
ALTERA


00 00 0002
SRAM 512KB
FLASH 4MB
SD CARD
DE2
ALTERA

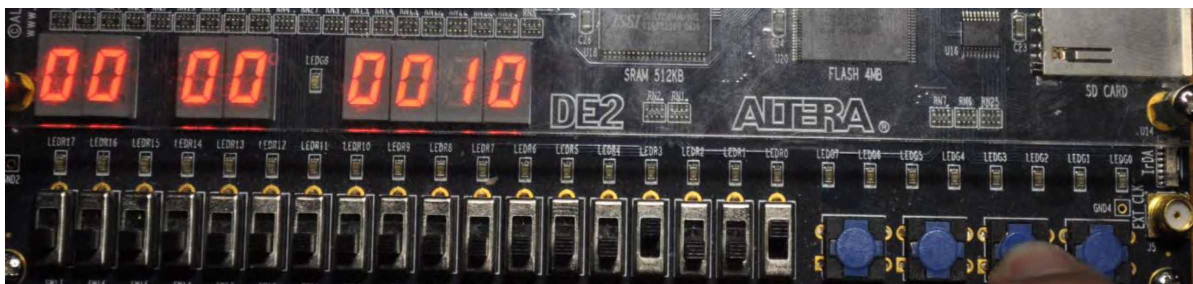

00 00 0004
SRAM 512KB
FLASH 4MB
SD CARD
DE2
ALTERA

00 00 0008
SRAM 512KB
FLASH 4MB
SD CARD
DE2
ALTERA


00 00 0010
SRAM 512KB
FLASH 4MB
SD CARD
DE2
ALTERA

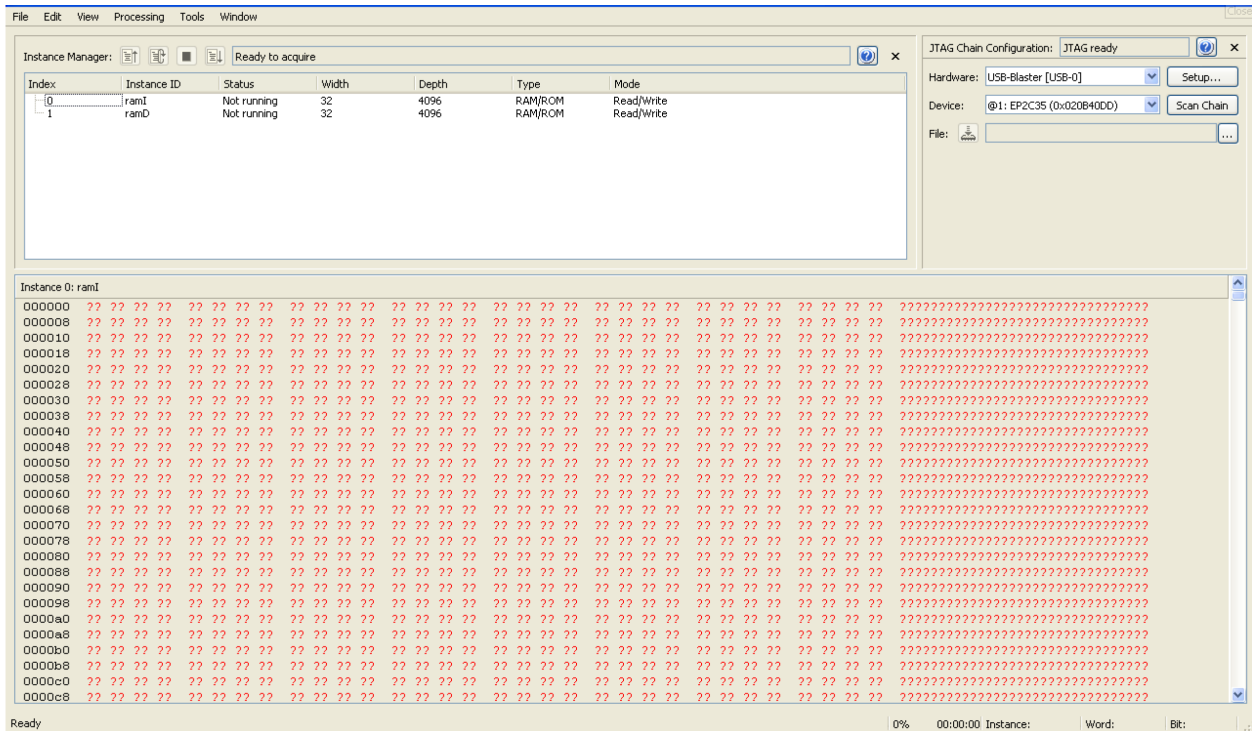


Το επόμενο παράδειγμα που μπορούμε να τρέξουμε είναι η πρόσθεση δύο αριθμών που ενεργοποιείτε από το opcode 010. Έτσι αποθηκεύουμε τον αριθμό 7 στην είσοδο Α και τον αριθμό 9 στην είσοδο Β. Ενεργοποιώντας συνεχόμενα το opcode 010 βλέπουμε το αποτέλεσμα στο δεκαεξαδικό x0010 που είναι και το αναμενόμενο. Τα αποτελέσματα φαίνονται στην παρακάτω εικόνα. Βέβαια επαληθεύσαμε την ALU με πολλά άλλα παραδείγματα για να είμαστε σίγουροι για την ορθή λειτουργία της.

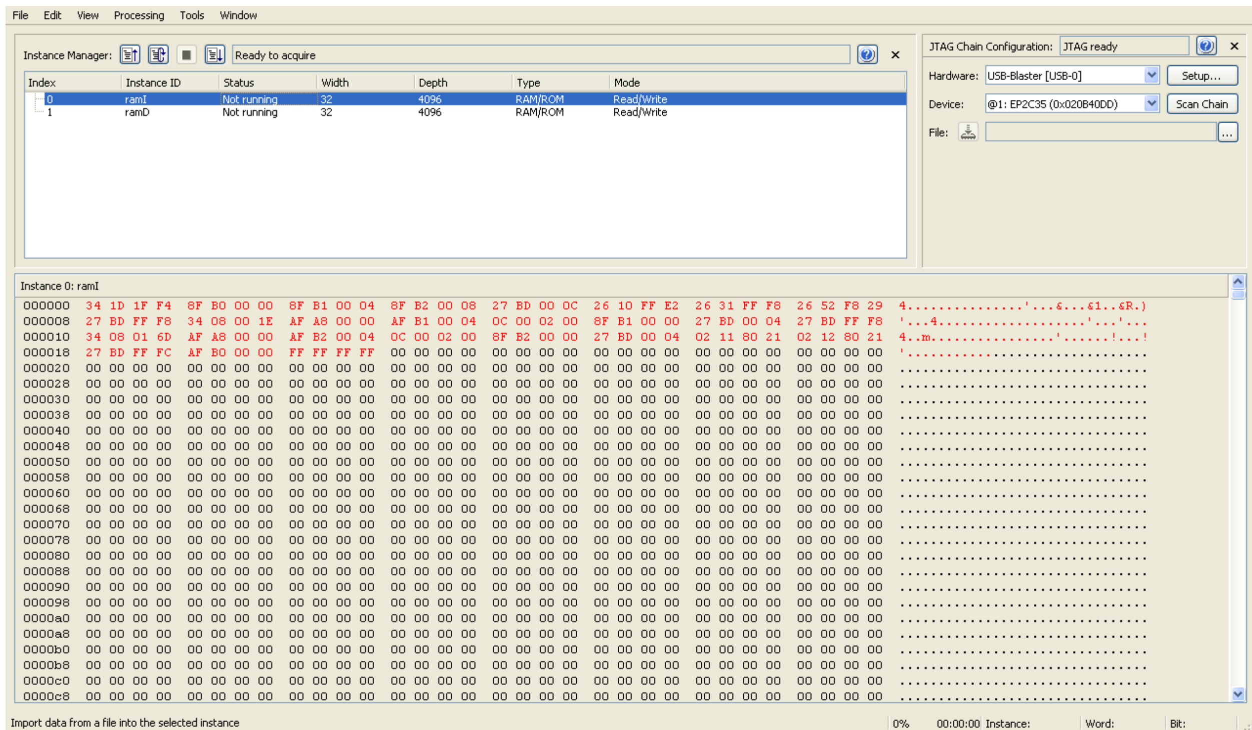


Σχ. 7.2: Πρόσθεση από το κύκλωμα της ALU των αριθμών 7 και 9 έχοντας ενεργοποιήσει το opcode 010. Το αποτέλεσμα είναι σε δεκαεξαδικό το x0010.

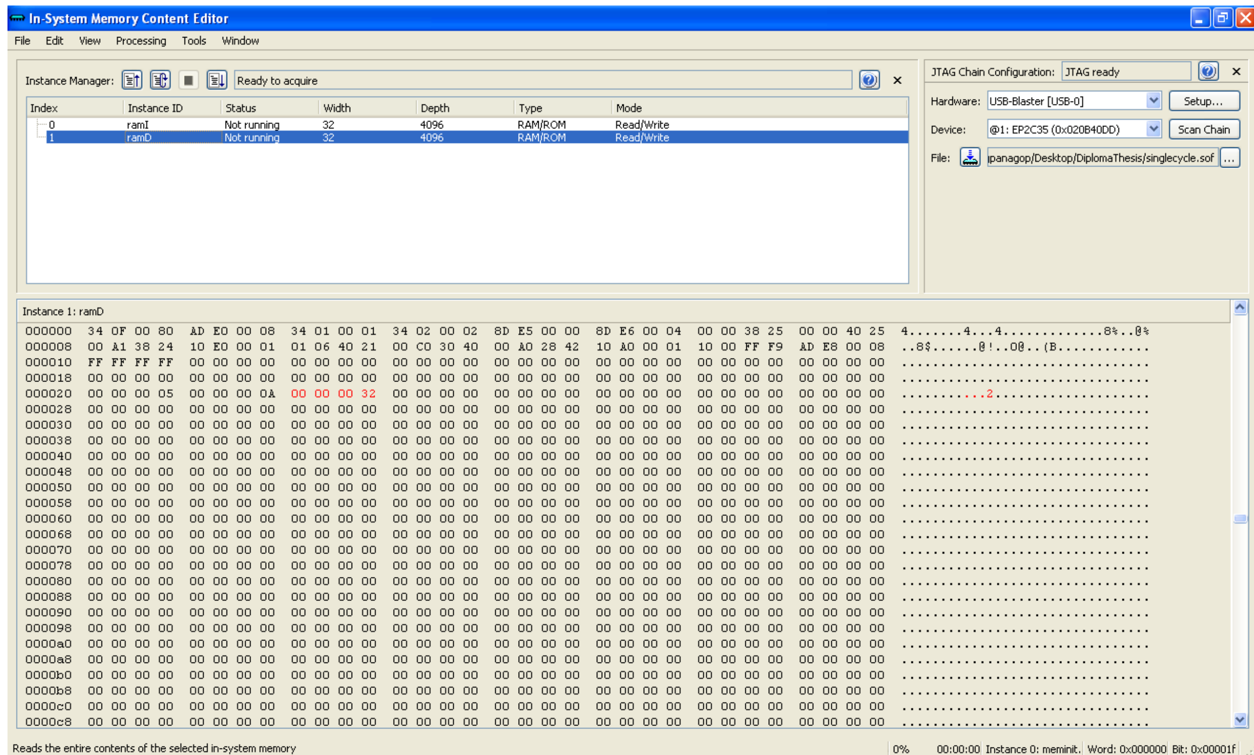
Έχουμε υλοποιήσει τα ίδια test και για τα άλλα υποκυκλώματα του επεξεργαστή όπως το register file και το control unit. Στις επόμενες παραγράφους παρουσιάζουμε τη λειτουργικότητα του επεξεργαστή και την εκτέλεση του προγράμματος multiplication.



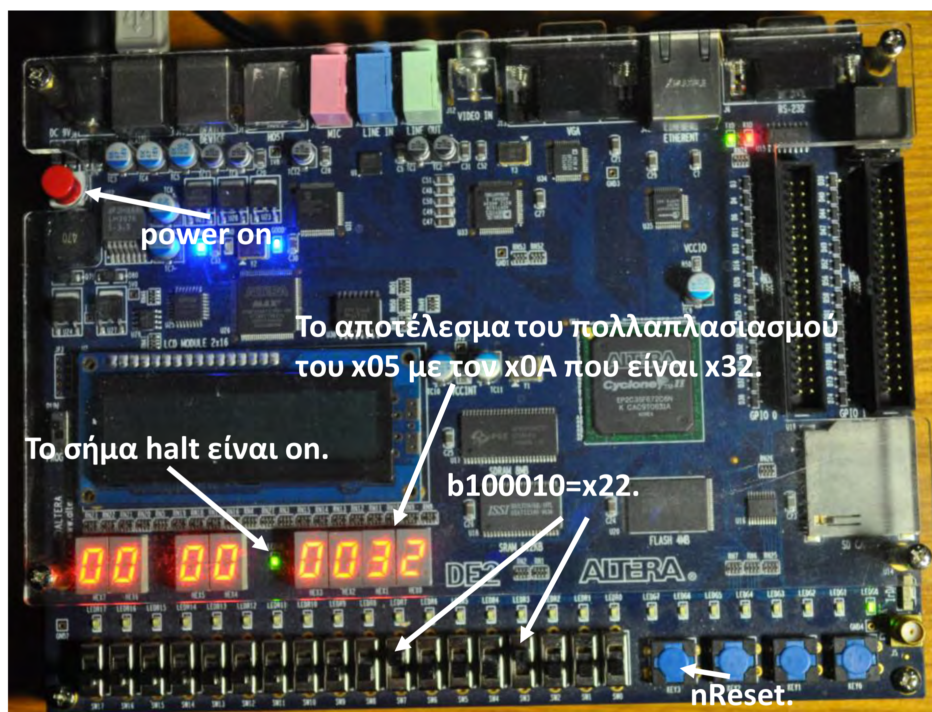
Σχ. 7.3: Περιεχόμενα της μνήμης ramI και ramD πριν την φόρτωση και εκτέλεση του προγράμματος από τον επεξεργαστή single-cycle.



Σχ. 7.4: Φόρτωση των εντολών από το αρχείο meminit.hex στη μνήμη ramI. Όταν αφήσουμε το reset ο επεξεργαστής θα αρχίσει να τρέχει και το πρόγραμμα θα εκτελεστεί.



Σχ. 7.5: Το αποτέλεσμα του πολλαπλασιασμού των αριθμών x05 και x0A είναι το x32 και είναι με κόκκινο στον editor των περιεχομένων της μνήμης του επεξεργαστή που είναι υλοποιημένος στο FPGA. Αυτό αποδεικνύει την ορθότητα της λειτουργίας του επεξεργαστή.



Σχ. 7.6: Το αποτέλεσμα του πολλαπλασιαμού στο FPGA.

8. Αποτελέσματα & Απόδοση των CPU

Σε αυτή την ενότητα θα κάνουμε μια ανασκόπηση των τριών υλοποιήσεων που παρουσιάστηκαν στα Κεφ. 4, 5 και 6 και θα κάνουμε μια σύγκριση από πολλές οπτικές γωνίες.

Πίν. 1: Κυκλωματικά στοιχεία που χρησιμοποιήθηκαν από το FPGA για την υλοποίηση των επεξεργαστών pipeline και dualcore και η συγκρισή τους.

	Pipelined CPU with Cache	Dual Core CPU
Total logic elements	5655	12210
Total registers	3611	7082
Operating frequency (MHz)	31.71	12.66
Critical path delay (ns)	31.532	79.012

Πίν. 2: Μέσος χρόνος διεκπεραίωσης και latency των επεξεργαστών pipeline και dualcore και η συγκρισή τους.

	Pipelined CPU with Cache	Dual Core CPU
Average cycles per instruction	1.69	1.19
Average instructions per cycle	0.59	0.84
Average instruction latency (ns)	266.76	471.31
Average time per instruction (ns)	53.35	94.26

Α. Βασικές κυκλωματικές δομές σε VHDL

Η VHDL είναι μια γλώσσα περιγραφής ψηφιακών κυκλωμάτων και χρησιμοποιείται για την *προσομοίωση* και *σύνθεση* ηλεκτρονικών κυκλωμάτων. Σύνθεση είναι η διαδικασία κατά την οποία ο κώδικας VHDL που περιγράφει το κύκλωμα μεταγλωττίζεται (compilation) και μετατρέπεται (mapped) σε μια τεχνολογική υλοποίηση όπως FPGA ή ASIC. Συνήθως πολλοί παροχείς FPGA (vendors) παρέχουν εργαλεία που κάνουν σύνθεση τη VHDL για να χρησιμοποιηθεί με τα chips τους. Αντιθέτως, τα εργαλεία για υλοποιήσεις ASIC έχουν πολύ υψηλό κόστος και είναι χρονοβόρα.

Είναι αρκετά σημαντικό να έχουμε υπόψη μας ότι δεν μπορούν να γίνουν σύνθεση όλες οι δομές VHDL ακόμη και αν μεταγλωττίζονται σωστά. Χαρακτηριστικό παράδειγμα είναι ο κώδικας για τα testbench ή εκφράσεις «wait for 10 ns;». Ο λόγος ύπαρξης αυτών των δομών είναι για κάνουμε προσομοίωση και όχι υλοποίηση. Παρότι κάθε εργαλείο σύνθεσης έχει διαφορετικές δυνατότητες υπάρχει ένα κοινό υποσύνολο της VHLD που γίνεται σύνθεσης από όλα τα εργαλεία. Το IEEE 1076.6 [1] ορίζει ένα επίσημο υποσύνολο της VHDL και μέρος αυτών των δομών παρουσιάζονται στα επόμενα υποκεφάλαια.

Μερικά παραδείγματα κώδικα VHDL που χρησιμοποιήθηκαν σε τμήματα του κώδικα για την υλοποίηση του επεξεργαστή δίδονται παρακάτω:

Α.1 Πολυπλέκτης (MUX)

Κώδικες VHDL που μπορούν να μετατραπούν σε πολυπλέκτη είναι οι ακόλουθοι:

```
-- template 1:
X <= A when S = '1' else B;

-- template 2:
with S select
  X <= A when '1',
       B when others;

-- template 3:
process(A,B,S)
begin
  case S is
    when '1'     => X <= A;
    when others => X <= B;
  end case;
end process;


-- template 4:
process(A,B,S)
begin
  if S = '1' then
    X <= A;
  else
    X <= B;
  end if;
end process;


-- template 5 - 4:1 MUX, where S is a 2-bit std_logic_vector :
process(A,B,C,D,S)
begin
```

```
  case S is
    when "00"   => X <= A;
    when "01"   => X <= B;
    when "10"   => X <= C;
    when others => X <= D;
  end case;
end process;
```

A.2 D Flip-Flop

Παρότι υπάρχουν πολλές κατηγορίες flip-flop αυτό που χρησιμοποιείται ευρέως στα ψηφιακά κυκλώματα είναι το flip-flop τύπου D. Παρακάτω παρουσιάζονται τρεις διαφορετικοί τρόποι υλοποίησής του σε VHDL.

```
-- recommended DFF template:
process(CLK)
begin
  -- use falling_edge(CLK) to sample at the falling edge instead
  if rising_edge(CLK) then
    Q <= D;
  end if;
end process;


-- alternative DFF template:
process
begin
  wait until CLK='1';
  Q <= D;
end process;


-- alternative template expands the ''rising_edge'' function above:
process(CLK)
begin
   if CLK = '1' and CLK'event then--use rising edge, use  "if CLK = '0' and
CLK'event" instead for falling edge
      Q <= D;
   end if;
end process;
```

B. Αναφορές

[1] "*IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis*".

[2] D. A. Patterson, J. L. Hennessy, "*Computer Organization and Design, the hardware/software interface*", 3rd Ed., 2005.

[3] John F. Wakerly, "*Digital Design: Principles and Practices*", 3rd Ed., 1999.

[4] Morris M. Mano, "*Digital Design*", 3rd Ed., 2001.

[5] C. Hamacher, Z. Vranesic and S. Zaky, "*Computer Organization*", 2001.

[6] J. Nurmi, "*Processor Design: System on Chip Computing for ASICs and FPGAs*", 2010.