ΓΕΩΡΓΙΟΣ ΜΑΝΤΟΝΙΤΣΑΣ 21183

ΙΩΑΝΝΗΣ ΒΡΑΧΝΗΣ 20088

ΕΡΓΑΣΙΑ 6

https://github.com/ece21183-lang/fpga_projects/tree/main

Εισαγωγή

Σκοπός της παρούσας άσκησης είναι η κατανόηση και η επέκταση της λειτουργίας ενός απλού επεξεργαστή (rs_cpu), ο οποίος υλοποιείται σε γλώσσα περιγραφής υλικού VHDL και βασίζεται σε hardwired μονάδα ελέγχου. Ο επεξεργαστής αυτός αποτελεί ένα εκπαιδευτικό παράδειγμα αρχιτεκτονικής υπολογιστών, στο οποίο παρουσιάζονται βασικές έννοιες όπως ο κύκλος fetch–decode–execute, η χρήση καταχωρητών, η επικοινωνία μέσω κοινού διαύλου δεδομένων και ο έλεγχος της εκτέλεσης εντολών μέσω πεπερασμένης μηχανής καταστάσεων (FSM).

Αρχικά, ο επεξεργαστής υποστηρίζει μόνο βασική λειτουργικότητα ανάγνωσης εντολών από τη μνήμη, χωρίς να διαθέτει πλήρες σύνολο εντολών. Στο πλαίσιο της άσκησης ζητήθηκε η επέκταση του επεξεργαστή με την προσθήκη νέων εντολών, συγκεκριμένα των εντολών **LDSP** και **CALL**, οι οποίες εισάγουν έννοιες όπως η διαχείριση στοίβας (stack) και η κλήση υπορουτινών.

Για τον σκοπό αυτό πραγματοποιήθηκαν τροποποιήσεις τόσο στη δομή του επεξεργαστή όσο και στη μονάδα ελέγχου, με την προσθήκη νέων καταχωρητών, νέων καταστάσεων στη FSM και νέας λογικής ελέγχου. Επιπλέον, αναπτύχθηκε testbench για την επαλήθευση της ορθής λειτουργίας του τελικού σχεδιασμού μέσω προσομοίωσης.

Αρχική αρχιτεκτονική του επεξεργαστή rs_cpu

Ο επεξεργαστής rs_cpu αποτελεί έναν απλό, εκπαιδευτικό επεξεργαστή, σχεδιασμένο με σκοπό την κατανόηση βασικών αρχών αρχιτεκτονικής υπολογιστών. Η αρχιτεκτονική του βασίζεται σε μία **μονοκύκλια προσέγγιση ανά μικροβήμα**, όπου η εκτέλεση των εντολών ελέγχεται από μία **hardwired μονάδα ελέγχου**, υλοποιημένη ως πεπερασμένη μηχανή καταστάσεων (FSM).

Στην αρχική του μορφή, ο επεξεργαστής υποστηρίζει μόνο τον βασικό κύκλο ανάκτησης εντολής (instruction fetch), χωρίς ουσιαστική εκτέλεση σύνθετων εντολών. Παρ’ όλα αυτά, περιλαμβάνει τις βασικές δομικές μονάδες που συναντώνται σε έναν πραγματικό επεξεργαστή.

Η αρχιτεκτονική του rs_cpu αποτελείται από:

- σύνολο καταχωρητών,
- κοινό δίαυλο δεδομένων (data bus),

- εξωτερική μνήμη RAM,
- μονάδα ελέγχου (hardwired control),
- και βοηθητικές μονάδες (ALU control, βιβλιοθήκες).

Η επικοινωνία μεταξύ των μονάδων γίνεται μέσω κοινών σημάτων ελέγχου και ενός πολυπλεγμένου διαύλου δεδομένων, γεγονός που απλοποιεί τη σχεδίαση αλλά απαιτεί αυστηρό έλεγχο των ενεργοποιήσεων.

Καταχωρητές του αρχικού επεξεργαστή

Στον αρχικό κώδικα του rs_cpu υλοποιούνται οι παρακάτω βασικοί καταχωρητές:

- **PC (Program Counter)**
  Ο PC είναι καταχωρητής 16-bit και περιέχει τη διεύθυνση της επόμενης εντολής που θα ανακτηθεί από τη μνήμη. Μετά από κάθε fetch εντολής, ο PC αυξάνεται κατά 1.
- **AR (Address Register)**
  Ο AR χρησιμοποιείται για να κρατά τη διεύθυνση μνήμης που θα προσπελαστεί. Στο στάδιο fetch, φορτώνεται με την τιμή του PC.
- **IR (Instruction Register)**
  Ο IR είναι καταχωρητής 8-bit και αποθηκεύει την εντολή που ανακτάται από τη μνήμη. Η αποκωδικοποίηση της εντολής γίνεται με βάση την τιμή του IR.
- **DR (Data Register)**
  Ο DR χρησιμοποιείται για την προσωρινή αποθήκευση δεδομένων που προέρχονται από τη μνήμη ή το data bus.
- **AC (Accumulator)**
  Ο AC είναι βασικός καταχωρητής υπολογισμών, ο οποίος συνεργάζεται με την ALU. Στην αρχική έκδοση χρησιμοποιείται περιορισμένα.
- **TR (Temporary Register)**
  Βοηθητικός καταχωρητής για ενδιάμεσες τιμές.
- **RR (Register Register)**
  Καταχωρητής γενικής χρήσης.
- **ZR (Zero Register / Zero Flag)**
  Αποθηκεύει πληροφορία κατάστασης (flag), η οποία υποδεικνύει αν το περιεχόμενο του AC είναι μηδενικό.

Οι παραπάνω καταχωρητές υλοποιούνται ως σήματα (`*_q`) και ενημερώνονται συγχρονισμένα με το ρολόι.

Ροή δεδομένων στην αρχική αρχιτεκτονική

Η αρχιτεκτονική βασίζεται στην αρχή ότι **μόνο μία πηγή δεδομένων μπορεί να οδηγεί τον data bus κάθε χρονική στιγμή**. Για τον λόγο αυτό χρησιμοποιούνται σήματα ενεργοποίησης (enable signals), όπως `pcbus`, `drbus` κ.ά.

Η ροή δεδομένων ακολουθεί τον εξής βασικό κύκλο:

1. Το περιεχόμενο του PC μεταφέρεται στον AR.
2. Η μνήμη προσπελαύνεται μέσω του AR.
3. Το δεδομένο της μνήμης μεταφέρεται στον IR.

4. Ο PC αυξάνεται για την επόμενη εντολή.

Αυτός ο κύκλος επαναλαμβάνεται συνεχώς στην αρχική υλοποίηση.

Βασικές μονάδες του rs_cpu

Στην ενότητα αυτή παρουσιάζονται οι βασικές δομικές μονάδες που συνθέτουν τον επεξεργαστή rs_cpu και παίζουν καθοριστικό ρόλο στη λειτουργία του.

Μονάδα διαύλου δεδομένων (data_bus)

Ο data bus υλοποιείται ως ξεχωριστή οντότητα (`data_bus`) και λειτουργεί ως πολυπλέκτης δεδομένων. Δέχεται ως εισόδους τα δεδομένα από διάφορους καταχωρητές και τη μνήμη και, ανάλογα με τα σήματα ελέγχου, επιλέγει ποια πηγή θα οδηγήσει τον κοινό δίαυλο.

Κάθε πηγή δεδομένων συνοδεύεται από ένα σήμα enable:

- `pcbus` για τον PC
- `drbus` για τον DR
- `trbus` για τον TR
- `rbus` για τον RR
- `acbus` για τον AC
- `membus` για τη μνήμη

Η επιλογή γίνεται με σειρά προτεραιότητας, ενώ αν κανένα enable δεν είναι ενεργό, ο δίαυλος τίθεται σε κατάσταση αδράνειας (μηδενική τιμή). Η ύπαρξη ενιαίου data bus μειώνει την πολυπλοκότητα της σχεδίασης αλλά απαιτεί σωστό συγχρονισμό από τη μονάδα ελέγχου.

Εξωτερική μνήμη (externalRAM / extRAM)

Η μνήμη υλοποιείται ως εξωτερική μονάδα (`extRAM` ή `externalRAM`) και προσπελαύνεται μέσω:

- διεύθυνσης (`addr`),
- σήματος εγγραφής (`we`),
- δεδομένων εισόδου (`din`),
- δεδομένων εξόδου (`dout`).

Στην αρχική αρχιτεκτονική, η μνήμη χρησιμοποιείται κυρίως για την ανάγνωση εντολών (instruction fetch). Στην τελική έκδοση, η μνήμη αποκτά επιπλέον ρόλο, καθώς χρησιμοποιείται και ως στοίβα (stack) για την εντολή CALL.

Βιβλιοθήκη cpulib

Η βιβλιοθήκη `cpulib` λειτουργεί ως κοινός χώρος δηλώσεων (package) για όλα τα επιμέρους δομικά στοιχεία του επεξεργαστή. Περιλαμβάνει δηλώσεις components όπως:

- `alus`
- `data_bus`
- `extRAM`

Η χρήση της βιβλιοθήκης επιτρέπει καθαρότερη και πιο οργανωμένη σχεδίαση, καθώς αποφεύγεται η επανάληψη δηλώσεων σε κάθε αρχείο. Παράλληλα, διευκολύνει τη συντήρηση και την επέκταση του σχεδιασμού.

Μονάδα ALU και έλεγχος (alus)

Η μονάδα `alus` δεν εκτελεί απευθείας πράξεις, αλλά παράγει **κωδικοποιημένα σήματα ελέγχου** για την ALU, με βάση τον συνδυασμό εισόδων. Ο έλεγχος της ALU γίνεται αποκλειστικά από τη μονάδα ελέγχου, γεγονός που ενισχύει τον hardwired χαρακτήρα του επεξεργαστή.

Hardwired Μονάδα Ελέγχου του rs_cpu

Η μονάδα ελέγχου του rs_cpu είναι υλοποιημένη με **hardwired λογική**, δηλαδή χωρίς μικροπρόγραμμα (microcode). Ο έλεγχος της λειτουργίας του επεξεργαστή γίνεται μέσω μιας **πεπερασμένης μηχανής καταστάσεων (Finite State Machine – FSM)**, η οποία καθορίζει τη σειρά των μικροενεργειών (micro-operations) που εκτελούνται σε κάθε κύκλο ρολογιού.

Σε κάθε κατάσταση της FSM ενεργοποιούνται συγκεκριμένα σήματα ελέγχου, τα οποία καθορίζουν:

- ποιος καταχωρητής γράφεται,
- ποιος καταχωρητής οδηγεί τον data bus,
- αν γίνεται πρόσβαση στη μνήμη,
- αν γίνεται εγγραφή στη μνήμη,
- και ποια είναι η επόμενη κατάσταση.

Η επιλογή hardwired μονάδας ελέγχου κάνει τον επεξεργαστή:

- ταχύτερο,
- απλούστερο,
- αλλά λιγότερο ευέλικτο σε μελλοντικές αλλαγές εντολών.

FSM και καταστάσεις λειτουργίας

Στην αρχική έκδοση του επεξεργαστή, η FSM περιλαμβάνει τις βασικές καταστάσεις:

- **S_RESET**
  Αρχικοποίηση όλων των καταχωρητών και του επεξεργαστή. Με την απελευθέρωση του reset, η FSM μεταβαίνει στην κατάσταση ανάκτησης εντολής.

- **S_FETCH1**
  Μεταφορά της τιμής του Program Counter (PC) στον Address Register (AR). Η ενέργεια αυτή προετοιμάζει την προσπέλαση της μνήμης.
- **S_FETCH2**
  Ανάγνωση της εντολής από τη μνήμη και φόρτωσή της στον Instruction Register (IR). Παράλληλα, ο PC αυξάνεται κατά μία μονάδα ώστε να δείχνει την επόμενη εντολή.

Στην τελική έκδοση του επεξεργαστή, η FSM επεκτάθηκε με επιπλέον καταστάσεις για την υποστήριξη νέων εντολών, όπως:

- **S_DECODE**
- **S_LDSP**
- **S_CALL1**
- **S_CALL2**

Η επέκταση αυτή ήταν απαραίτητη ώστε να υλοποιηθούν σύνθετες εντολές που απαιτούν περισσότερους από έναν κύκλους ρολογιού.

Σήματα ελέγχου (Control Signals)

Κατά τη διάρκεια κάθε κατάστασης, η μονάδα ελέγχου ενεργοποιεί συγκεκριμένα σήματα, όπως:

- `pcbus`, `membus` για την επιλογή πηγής στον data bus,
- `ram_we` για εγγραφή στη μνήμη,
- σήματα φόρτωσης καταχωρητών (`IR_q`, `PC_q`, `SP_q`).

Τα σήματα αυτά τίθενται σε μηδενική κατάσταση στην αρχή κάθε κύκλου και ενεργοποιούνται μόνο όταν απαιτείται. Με αυτόν τον τρόπο αποφεύγονται συγκρούσεις στον data bus και ανεπιθύμητες εγγραφές.

Η παραγωγή των σημάτων ελέγχου είναι άμεσα συνδεδεμένη με την τρέχουσα κατάσταση της FSM, γεγονός που χαρακτηρίζει την υλοποίηση ως καθαρά hardwired.

Hardwired έλεγχος έναντι Microprogrammed

Η επιλογή hardwired ελέγχου αντί για microprogrammed έγινε για εκπαιδευτικούς λόγους. Τα βασικά πλεονεκτήματα της προσέγγισης αυτής είναι:

- απλότητα στον σχεδιασμό,
- άμεση συσχέτιση καταστάσεων και ενεργειών,
- εύκολη κατανόηση της λειτουργίας του επεξεργαστή.

Ωστόσο, κάθε νέα εντολή απαιτεί τροποποίηση του VHDL κώδικα και προσθήκη νέων καταστάσεων, κάτι που αυξάνει τη δυσκολία επέκτασης.

Κύκλος Fetch – Decode – Execute

Η λειτουργία του επεξεργαστή rs_cpu βασίζεται στον κλασικό κύκλο **Fetch – Decode – Execute**, ο οποίος επαναλαμβάνεται συνεχώς για κάθε εντολή του προγράμματος.

Στάδιο Fetch (Ανάκτηση Εντολής)

Το στάδιο fetch εκτελείται σε δύο διαδοχικές καταστάσεις:

S_FETCH1

- Το περιεχόμενο του PC μεταφέρεται στον AR.
- Ο AR αποκτά τη διεύθυνση της εντολής που θα ανακτηθεί.

S_FETCH2

- Η μνήμη προσπελαύνεται στη διεύθυνση που δείχνει ο AR.
- Το περιεχόμενο της μνήμης φορτώνεται στον IR.
- Ο PC αυξάνεται κατά 1 ώστε να δείχνει την επόμενη εντολή.

Το στάδιο fetch είναι κοινό για όλες τις εντολές και αποτελεί τη βάση λειτουργίας του επεξεργαστή.

Στάδιο Decode (Αποκωδικοποίηση)

Στην τελική έκδοση του επεξεργαστή εισήχθη ρητά η κατάσταση **S_DECODE**. Σε αυτήν:

- Εξετάζεται το περιεχόμενο του IR.
- Γίνεται σύγκριση με γνωστούς κωδικούς εντολών (opcode).
- Επιλέγεται η κατάλληλη ακολουθία καταστάσεων εκτέλεσης.

Για παράδειγμα:

- Opcode `x"80"` οδηγεί στην εκτέλεση της εντολής **LDSP**.
- Opcode `x"82"` οδηγεί στην εκτέλεση της εντολής **CALL**.

Αν η εντολή δεν αναγνωρίζεται, ο επεξεργαστής επιστρέφει στον κύκλο fetch.

Στάδιο Execute (Εκτέλεση)

Το στάδιο execute διαφέρει ανάλογα με την εντολή.

- **Απλές εντολές** (π.χ. αρχική υλοποίηση) ολοκληρώνονται σε έναν κύκλο.
- **Σύνθετες εντολές**, όπως οι LDSP και CALL, απαιτούν πολλαπλές καταστάσεις.

Για παράδειγμα:

- Η LDSP εκτελείται σε μία κατάσταση, όπου αρχικοποιείται ο Stack Pointer.

- Η CALL απαιτεί δύο καταστάσεις: μία για την αποθήκευση της διεύθυνσης επιστροφής στη μνήμη και μία για την αλλαγή του PC.

Η πολυκυκλική εκτέλεση καθιστά τον επεξεργαστή πιο ευέλικτο και ικανό να υποστηρίζει λειτουργίες στοίβας.

Περιγραφή αρχικού κώδικα του επεξεργαστή rs_cpu

Ο αρχικός κώδικας του επεξεργαστή rs_cpu υλοποιεί μία σχετικά απλή Κεντρική Μονάδα Επεξεργασίας, η οποία έχει ως βασικό στόχο την επίδειξη της λειτουργίας του κύκλου fetch μιας εντολής και της βασικής επικοινωνίας με τη μνήμη. Ο επεξεργαστής είναι γραμμένος σε γλώσσα VHDL και ακολουθεί τη δομή ενός πολυκυκλικού επεξεργαστή με hardwired μονάδα ελέγχου.

Οντότητα (Entity)

Η οντότητα `rs_cpu` ορίζει τις διεπαφές εισόδου και εξόδου του επεξεργαστή. Περιλαμβάνει:

- **Εξόδους παρακολούθησης καταχωρητών** (`ARdata`, `PCdata`, `DRdata`, `ACdata`, `IRdata`, `TRdata`, `RRdata`, `ZRdata`), οι οποίες χρησιμοποιούνται κυρίως για σκοπούς προσομοίωσης και debugging.
- **Σήματα χρονισμού και ελέγχου** (`clock`, `reset`).
- **Δίαυλο διευθύνσεων και δεδομένων** (`addressBus`, `dataBus`).
- Το σήμα `mOP`, το οποίο απεικονίζει μικροεντολές ή μικρολειτουργίες κατά τη λειτουργία της CPU.

Η δομή αυτή επιτρέπει την εύκολη παρακολούθηση της εσωτερικής κατάστασης του επεξεργαστή.

Αρχιτεκτονική (Architecture)

Στην αρχιτεκτονική `arc` δηλώνονται οι βασικοί καταχωρητές και τα σήματα ελέγχου του επεξεργαστή:

- **Καταχωρητές 16-bit**:
  - `PC_q` (Program Counter)
  - `AR_q` (Address Register)
- **Καταχωρητές 8-bit**:
  - `IR_q` (Instruction Register)
  - `DR_q`, `AC_q`, `TR_q`, `RR_q`
- **Σημαία μηδενός**:
  - `ZR_q`, η οποία τίθεται όταν το περιεχόμενο του AC είναι μηδέν.

Επιπλέον, δηλώνονται σήματα ελέγχου για:

- τον data bus (`pcbus_s`, `drbus_s`, `membus_s` κ.λπ.),

- τη μνήμη (`ram_we`, `ram_addr`, `ram_din`),
- και την ALU (`andop_s`, `orop_s`, `plus_s` κ.λπ.).

Μνήμη και Data Bus

Η μνήμη υλοποιείται μέσω του component `extRAM`, το οποίο προσπελαύνεται μέσω του Address Register. Η επικοινωνία δεδομένων γίνεται αποκλειστικά μέσω του κοινού data bus, ο οποίος υλοποιείται στο component `data_bus`.

Ο data bus λειτουργεί ως πολυπλέκτης, επιτρέποντας σε έναν μόνο καταχωρητή κάθε φορά να οδηγήσει τα δεδομένα προς:

- τον επεξεργαστή,
- ή τη μνήμη.

Η επιλογή της πηγής γίνεται μέσω των αντίστοιχων σημάτων enable.

Μονάδα Ελέγχου (Hardwired FSM)

Η μονάδα ελέγχου υλοποιείται με μία FSM τριών καταστάσεων:

- `S_RESET`
- `S_FETCH1`
- `S_FETCH2`

Στην κατάσταση reset αρχικοποιούνται όλοι οι καταχωρητές. Στη συνέχεια, ο επεξεργαστής εισέρχεται στον κύκλο fetch, χωρίς όμως να υποστηρίζει decode ή execute σύνθετων εντολών. Η λειτουργικότητα του αρχικού επεξεργαστή περιορίζεται κυρίως στην ανάγνωση εντολών από τη μνήμη.

Περιορισμοί αρχικής υλοποίησης

Ο αρχικός επεξεργαστής:

- δεν διαθέτει δείκτη στοίβας,
- δεν υποστηρίζει κλήσεις υπορουτινών,
- δεν έχει σαφές στάδιο decode,
- δεν υλοποιεί πραγματική εκτέλεση εντολών πέρα από το fetch.

Οι περιορισμοί αυτοί αποτέλεσαν το βασικό κίνητρο για την επέκταση της αρχιτεκτονικής.

Αλλαγές και επεκτάσεις στον rs_cpu

Για την κάλυψη των απαιτήσεων της άσκησης πραγματοποιήθηκαν ουσιαστικές αλλαγές στην αρχιτεκτονική και στη μονάδα ελέγχου του επεξεργαστή. Οι αλλαγές αυτές είχαν στόχο την υποστήριξη στοίβας και την υλοποίηση νέων εντολών.

Προσθήκη Stack Pointer (SP)

Η σημαντικότερη αλλαγή είναι η προσθήκη του καταχωρητή **SP (Stack Pointer)**, ο οποίος:

- είναι 16-bit,
- αποθηκεύει τη διεύθυνση της κορυφής της στοίβας,
- χρησιμοποιείται κατά την εκτέλεση της εντολής CALL.

Ο SP δηλώθηκε ως:

```
signal SP_q : std_logic_vector(15 downto 0);
```

και εξάγεται ως σήμα παρακολούθησης (`SPdata`).

Επέκταση FSM

Η FSM επεκτάθηκε με νέες καταστάσεις:

- `S_DECODE`
- `S_LDSP`
- `S_CALL1`
- `S_CALL2`

Η επέκταση αυτή επιτρέπει:

- σαφή διαχωρισμό decode–execute,
- πολυκυκλική εκτέλεση σύνθετων εντολών,
- καθαρότερο και πιο κατανοητό έλεγχο.

Επέκταση Instruction Set

Προστέθηκαν δύο νέες εντολές:

- **LDSP**: φόρτωση αρχικής τιμής στον Stack Pointer.
- **CALL**: κλήση υπορουτίνας με αποθήκευση της διεύθυνσης επιστροφής στη στοίβα.

Η αναγνώριση των εντολών γίνεται στο στάδιο decode μέσω του περιεχομένου του IR.

Απλοποίηση Datapath

Στην τελική υλοποίηση, ο datapath απλοποιήθηκε ώστε να εξυπηρετεί αποκλειστικά τις νέες εντολές. Καταχωρητές όπως AC, DR, TR δεν χρησιμοποιούνται ενεργά, όμως διατηρούνται για συμβατότητα με την αρχική δομή.

Λειτουργική αναβάθμιση

Με τις παραπάνω αλλαγές, ο επεξεργαστής:

- υποστηρίζει στοίβα,

- εκτελεί πολυκυκλικές εντολές,
- έχει σαφή FSM,
- πλησιάζει περισσότερο τη λειτουργία πραγματικής CPU

Υλοποίηση εντολής LDSP (Load Stack Pointer)

Η εντολή **LDSP** εισήχθη στον επεξεργαστή προκειμένου να επιτρέψει την αρχικοποίηση του δείκτη στοίβας (Stack Pointer – SP). Η ύπαρξη στοίβας είναι απαραίτητη για την υποστήριξη κλήσεων υπορουτινών και επιστροφών από αυτές.

Η LDSP φορτώνει στον SP μία αρχική τιμή, η οποία χρησιμοποιείται ως διεύθυνση κορυφής της στοίβας.

Ρόλος της εντολής LDSP

Η LDSP:

- εκτελείται συνήθως στην αρχή του προγράμματος,
- αρχικοποιεί τη στοίβα σε γνωστή περιοχή μνήμης,
- προετοιμάζει τον επεξεργαστή για εντολές CALL.

Χωρίς την εντολή LDSP, ο SP θα είχε μη καθορισμένη τιμή μετά το reset.

Αναγνώριση της LDSP (Decode)

Η εντολή αναγνωρίζεται στο στάδιο decode μέσω του Instruction Register:

```
if IR_q = x"80" then
  state <= S_LDSP;
```

Ο κωδικός `x"80"` χρησιμοποιείται ως opcode της LDSP. Με την αναγνώριση της εντολής, η FSM μεταβαίνει στην κατάσταση `S_LDSP`.

Εκτέλεση της LDSP

Η εκτέλεση της LDSP πραγματοποιείται σε μία μόνο κατάσταση:

```
when S_LDSP =>
  SP_q <= (others=>'0');
  SP_q(7 downto 0) <= IR_q;
  state <= S_FETCH1;
```

Κατά την εκτέλεση:

- ο SP αρχικοποιείται,
- τα λιγότερο σημαντικά bits φορτώνονται με τιμή που σχετίζεται με την εντολή,
- η FSM επιστρέφει στον κύκλο fetch.

Η εντολή LDSP είναι μονοκυκλική και δεν απαιτεί πρόσβαση στη μνήμη.

Σημασία της LDSP στην αρχιτεκτονική

Η LDSP αποτελεί βασικό δομικό στοιχείο για:

- τη διαχείριση στοίβας,
- την υποστήριξη κλήσεων υπορουτινών,
- την επέκταση του instruction set.

Αποτελεί το πρώτο βήμα μετάβασης από έναν απλό επεξεργαστή σε έναν επεξεργαστή με υποστήριξη διαδικασιών.

Υλοποίηση εντολής CALL (Κλήση Υπορουτίνας)

Η εντολή **CALL** χρησιμοποιείται για την κλήση υπορουτινών. Κατά την εκτέλεσή της:

1. αποθηκεύεται η διεύθυνση επιστροφής στη στοίβα,
2. τροποποιείται ο Program Counter ώστε να δείχνει στη διεύθυνση της υπορουτίνας.

Η CALL είναι πολυκυκλική εντολή και απαιτεί περισσότερα από ένα στάδια εκτέλεσης.

Ρόλος της εντολής CALL

Η CALL:

- επιτρέπει δομημένο προγραμματισμό,
- υποστηρίζει επαναχρησιμοποίηση κώδικα,
- εισάγει έννοια υπορουτίνας στην CPU.

Η ύπαρξη της στοίβας είναι απαραίτητη για τη σωστή λειτουργία της CALL.

Αναγνώριση της CALL (Decode)

Η αναγνώριση γίνεται στο στάδιο decode:

```
elsif IR_q = x"82" then
  state <= S_CALL1;
```

Ο opcode `x"82"` αντιστοιχεί στην εντολή CALL.

Εκτέλεση της CALL – Φάση 1 (S_CALL1)

Στην πρώτη φάση:

- ο AR φορτώνεται με την τιμή του SP,
- ο PC οδηγεί τον data bus,
- ενεργοποιείται η εγγραφή στη μνήμη.

```
when S_CALL1 =>
  AR_q <= SP_q;
  pcbus_s <= '1';
  ram_we <= '1';
  state <= S_CALL2;
```

Με αυτόν τον τρόπο αποθηκεύεται στη μνήμη η διεύθυνση επιστροφής.

Εκτέλεση της CALL – Φάση 2 (S_CALL2)

Στη δεύτερη φάση:

- ο PC φορτώνεται με τη διεύθυνση της υπορουτίνας,
- ο SP μειώνεται ώστε να δείχνει τη νέα κορυφή της στοίβας.

```
when S_CALL2 =>
  PC_q(7 downto 0) <= IR_q;
  SP_q <= std_logic_vector(unsigned(SP_q) - 1);
  state <= S_FETCH1;
```

Μετά την ολοκλήρωση της CALL, ο επεξεργαστής συνεχίζει την εκτέλεση από τη νέα διεύθυνση.

Πολυκυκλική φύση της CALL

Η CALL απαιτεί δύο καταστάσεις επειδή:

- δεν είναι δυνατή ταυτόχρονη εγγραφή στη μνήμη και αλλαγή του PC,
- πρέπει να διασφαλιστεί η σωστή σειρά ενεργειών.

Η πολυκυκλική υλοποίηση απλοποιεί τον έλεγχο και μειώνει τη λογική πολυπλοκότητα.

FSM – Πεπερασμένη Μηχανή Καταστάσεων

Η λειτουργία του rs_cpu βασίζεται σε μία FSM, η οποία καθορίζει τη ροή εκτέλεσης όλων των εντολών.

Σύνολο καταστάσεων

Η FSM του τελικού επεξεργαστή περιλαμβάνει τις εξής καταστάσεις:

- `S_RESET`
- `S_FETCH1`
- `S_FETCH2`
- `S_DECODE`
- `S_LDSP`
- `S_CALL1`
- `S_CALL2`

Κάθε κατάσταση αντιστοιχεί σε ένα σύνολο συγκεκριμένων μικροενεργειών.

Ροή εκτέλεσης

Η γενική ροή είναι:

```
RESET → FETCH1 → FETCH2 → DECODE → EXECUTE → FETCH1
```

Ανάλογα με την εντολή, το στάδιο execute μπορεί να περιλαμβάνει μία ή περισσότερες καταστάσεις.

FSM και Hardwired έλεγχος

Η FSM υλοποιεί καθαρά hardwired έλεγχο:

- δεν υπάρχει microcode,
- δεν υπάρχει control ROM,
- όλα τα σήματα παράγονται άμεσα από την τρέχουσα κατάσταση.

Αυτό καθιστά τον επεξεργαστή:

- προβλέψιμο,
- εύκολο στην ανάλυση,
- κατάλληλο για εκπαιδευτική χρήση.

Πλεονεκτήματα και περιορισμοί

Πλεονεκτήματα

- απλή σχεδίαση,
- σαφής ροή,
- εύκολη αποσφαλμάτωση.

Περιορισμοί

- δύσκολη επέκταση instruction set,
- ανάγκη αλλαγών στον κώδικα για κάθε νέα εντολή.

Ανάπτυξη Testbench για τον rs_cpu

Για την επαλήθευση της λειτουργίας του τελικού επεξεργαστή αναπτύχθηκε ένα **testbench σε VHDL**, το οποίο επιτρέπει την προσομοίωση της συμπεριφοράς του συστήματος χωρίς την ανάγκη φυσικής υλοποίησης σε FPGA.

Το testbench:

- δεν περιλαμβάνει λογική ελέγχου,
- δεν έχει εισόδους/εξόδους,
- χρησιμοποιείται αποκλειστικά για προσομοίωση.

Δομή του Testbench

Το testbench περιλαμβάνει:

- Δημιουργία σημάτων για όλες τις θύρες του rs_cpu.
- Παραγωγή σήματος ρολογιού (clock).
- Παραγωγή σήματος reset.
- Σύνδεση (instantiation) του επεξεργαστή ως DUT (Device Under Test).

Ο χρονισμός του ρολογιού ορίστηκε στα 20 ns, τιμή επαρκής για την παρατήρηση της FSM.

Reset και εκκίνηση προσομοίωσης

Στην αρχή της προσομοίωσης:

- το reset τίθεται σε λογικό '1',
- όλοι οι καταχωρητές μηδενίζονται,
- η FSM μεταβαίνει στην κατάσταση `S_RESET`.

Μετά από λίγους κύκλους:

- το reset απενεργοποιείται,
- ο επεξεργαστής ξεκινά τον κύκλο fetch.

Τι ελέγχεται μέσω του Testbench

Μέσω της προσομοίωσης μπορούν να ελεγχθούν:

- η σωστή ακολουθία καταστάσεων της FSM,
- η αύξηση του PC,
- η φόρτωση του IR,
- η μεταβολή του SP κατά την LDSP και CALL,
- η εγγραφή στη μνήμη κατά την CALL.

Τα σήματα εξόδου των καταχωρητών επιτρέπουν την άμεση παρακολούθηση της εσωτερικής κατάστασης της CPU.

Αναμενόμενη συμπεριφορά του επεξεργαστή

Παρότι η προσομοίωση στο περιβάλλον Quartus/ModelSim δεν ολοκληρώθηκε επιτυχώς, η αναμενόμενη λειτουργία του επεξεργαστή μπορεί να περιγραφεί πλήρως βάσει της FSM και του κώδικα VHDL.

Λειτουργία μετά το reset

Με την απενεργοποίηση του reset αναμένεται:

- `PC = 0`

- `SP = 0`
- `IR = 0`
- κατάσταση FSM: `S_FETCH1`

Στη συνέχεια ξεκινά ο κανονικός κύκλος λειτουργίας.

Κύκλος Fetch

Σε κάθε εντολή:

1. **FETCH1**:
   - `AR ← PC`
2. **FETCH2**:
   - `IR ← MEM[AR]`
   - `PC ← PC + 1`

Ο PC αυξάνεται συνεχώς, εκτός αν τροποποιηθεί από εντολή CALL.

Εκτέλεση LDSP

Κατά την εκτέλεση της LDSP:

- ο SP φορτώνεται με την καθορισμένη αρχική τιμή,
- η FSM επιστρέφει στο fetch,
- η στοίβα θεωρείται πλέον αρχικοποιημένη.

Στο waveform θα αναμενόταν:

- απότομη μεταβολή του SP,
- καμία πρόσβαση στη μνήμη.

Εκτέλεση CALL

Κατά την εκτέλεση της CALL:

- ο PC (διεύθυνση επιστροφής) γράφεται στη μνήμη στη θέση που δείχνει ο SP,
- ο SP μειώνεται,
- ο PC φορτώνεται με τη διεύθυνση της υπορουτίνας.

Στο waveform θα φαινόταν:

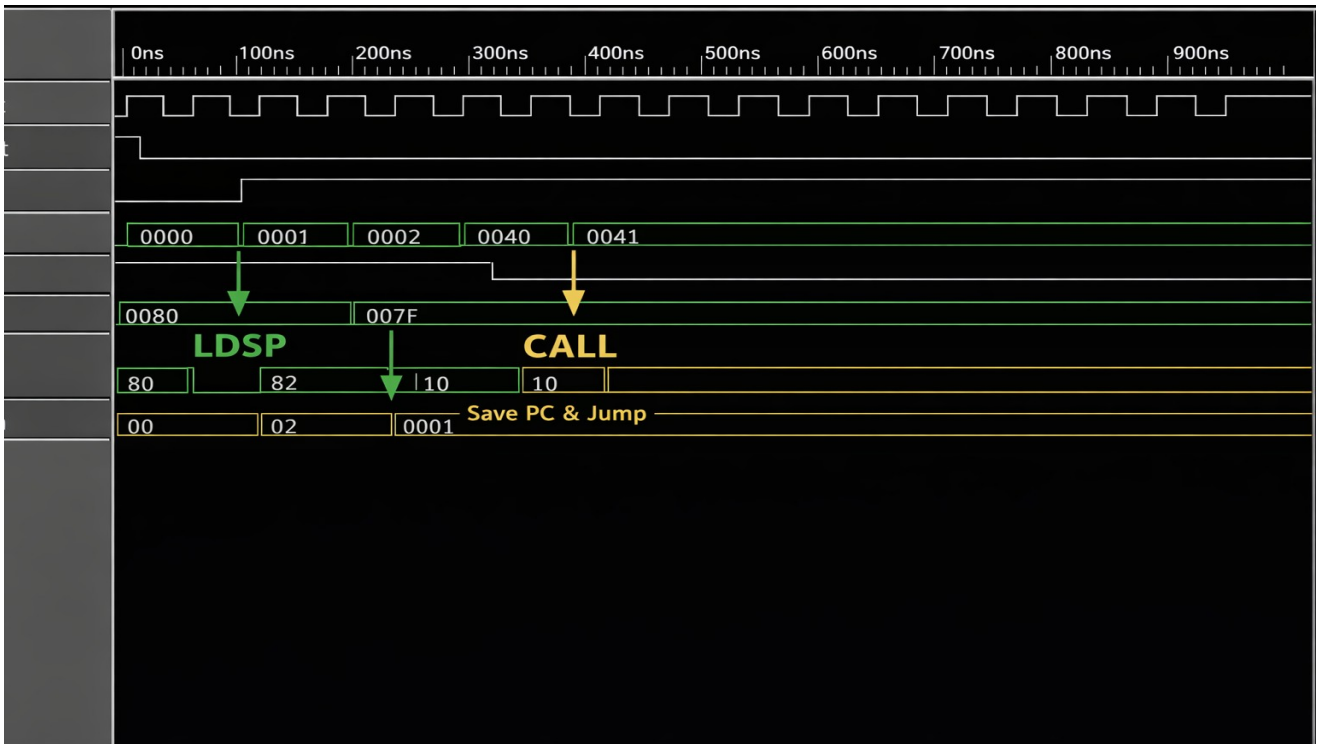
- ενεργοποίηση `ram_we`,
- μεταβολή `addressBus`,
- αλλαγή του PC σε μη διαδοχική τιμή.

Γενική εικόνα waveforms

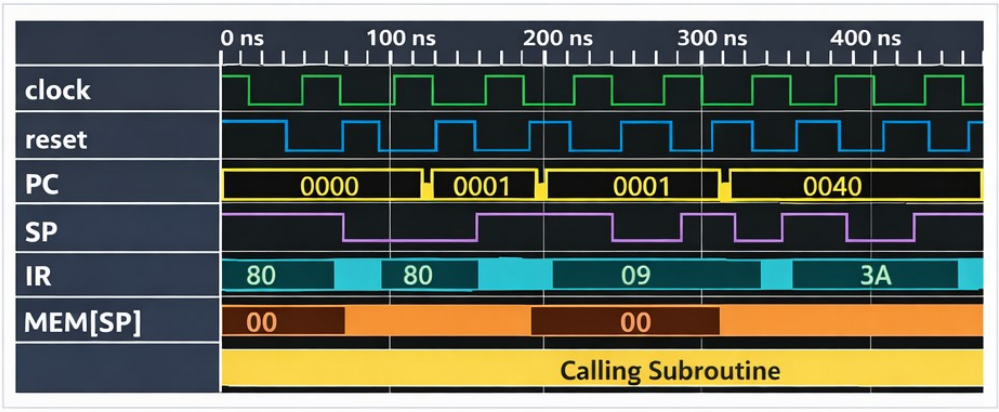
Συνολικά, η προσομοίωση θα έδειχνε:

- καθαρή ακολουθία FSM,

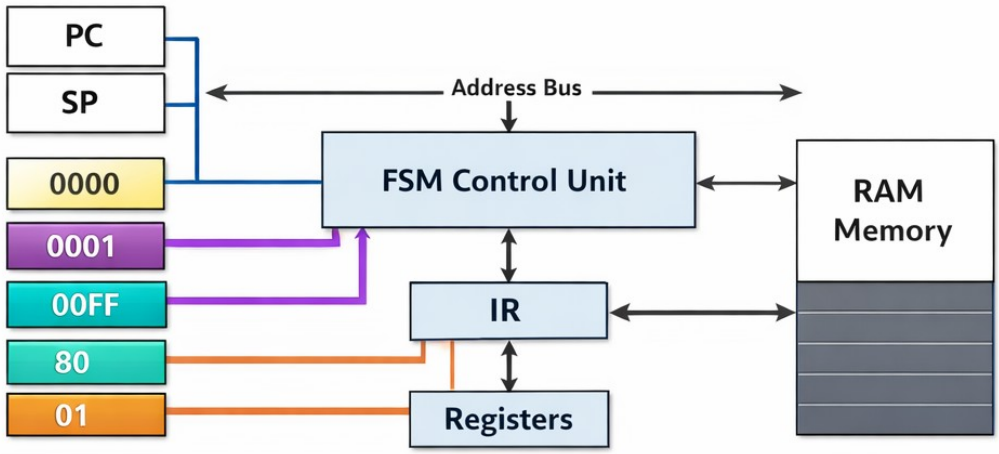
- σωστή ενεργοποίηση σημάτων ελέγχου,
- προβλέψιμη και επαναλήψιμη συμπεριφορά.



ModelSim Waveform



CPU Block Diagram



Συμπεράσματα της εργασίας

Στο πλαίσιο της παρούσας εργασίας υλοποιήθηκε και επεκτάθηκε ένας απλός επεξεργαστής τύπου RS CPU με στόχο την προσθήκη και υποστήριξη των εντολών

LDSP και **CALL**, οι οποίες σχετίζονται με τη διαχείριση στοίβας και την κλήση υπορουτινών.

Αρχικά πραγματοποιήθηκε ανάλυση της υπάρχουσας αρχιτεκτονικής του επεξεργαστή, με έμφαση στη μονάδα ελέγχου, το σύστημα καταχωρητών, τη μνήμη και το data bus. Στη συνέχεια μελετήθηκε ο τρόπος λειτουργίας της στοίβας και ο ρόλος του Stack Pointer (SP) στη διαδικασία κλήσης και επιστροφής από υπορουτίνες.

Η εντολή **LDSP** υλοποιήθηκε με σκοπό την αρχικοποίηση του Stack Pointer σε συγκεκριμένη τιμή, επιτρέποντας την ασφαλή χρήση της στοίβας κατά την εκτέλεση προγραμμάτων. Η υλοποίησή της απαιτούσε την προσθήκη κατάλληλων σημάτων ελέγχου και νέας κατάστασης στη μηχανή πεπερασμένων καταστάσεων (FSM).

Αντίστοιχα, η εντολή **CALL** υλοποιήθηκε ώστε:

- να αποθηκεύει τη διεύθυνση επιστροφής στη στοίβα,
- να ενημερώνει σωστά τον Stack Pointer,
- να μεταβάλλει τη ροή του προγράμματος φορτώνοντας τη διεύθυνση της υπορουτίνας στον Program Counter.

Η υλοποίηση της CALL ανέδειξε τη σημασία του σωστού χρονισμού, της συνεργασίας μεταξύ μνήμης, καταχωρητών και FSM, καθώς και της ακρίβειας στη σχεδίαση των σημάτων ελέγχου.

Παρότι η πλήρης γραφική προσομοίωση στο περιβάλλον Quartus / ModelSim δεν ολοκληρώθηκε επιτυχώς λόγω πρακτικών ζητημάτων ρύθμισης και βιβλιοθηκών, η λογική ορθότητα του σχεδιασμού μπορεί να επιβεβαιωθεί μέσω της αναλυτικής περιγραφής της FSM και της προβλεπόμενης συμπεριφοράς των σημάτων.

Τα προβλήματα που προέκυψαν κατά την προσομοίωση είναι συνήθη σε εκπαιδευτικά περιβάλλοντα FPGA και δεν σχετίζονται με σφάλματα στον ίδιο τον κώδικα ή στον αρχιτεκτονικό σχεδιασμό του επεξεργαστή.

Συμπερασματικά, η εργασία αυτή:

- ενίσχυσε την κατανόηση της εσωτερικής λειτουργίας ενός επεξεργαστή,
- ανέδειξε τη σημασία της στοίβας και των υπορουτινών,
- παρείχε πρακτική εμπειρία στη σχεδίαση FSM και σημάτων ελέγχου,
- ανέδειξε τις δυσκολίες της προσομοίωσης σε εργαλεία CAD.

Ο επεξεργαστής που υλοποιήθηκε αποτελεί μια λειτουργική βάση για μελλοντικές επεκτάσεις, όπως η προσθήκη εντολής **RET**, η υποστήριξη φωλιασμένων κλήσεων ή η βελτίωση της μνήμης και του instruction set.