קורס ארכיטקטורה של מחשבים

פרויקט: סימולטור למעבד מרובה ליבות

מגישים:

דין כרמון, 209325026

נדב מלאכי, 207386624

שיר גיאת, 315523639

תוכן עניינים

1. מבוא

במסגרת קורס ארכיטקטורה של מחשבים, פיתחנו סימולטור מתקדם למעבד מרובה ליבות. הסימולטור מדמה מערכת בעלת ארבע ליבות עיבוד עצמאיות, המתקשרות ביניהן באמצעות באס משותף. כל ליבה כוללת צנרת בת חמישה שלבים, זיכרון הוראות פרטי ומטמון נתונים. הסימולטור מיישם פרוטוקול קוהרנטיות מסוג MESI לניהול המטמונים, ומאפשר גישה יעילה לזיכרון ראשי משותף.

המטרה המרכזית של הפרויקט היא לספק סביבת סימולציה מדויקת המאפשרת חקירה מעמיקה של האתגרים בתכנון מעבד מודרני מרובה ליבות. הסימולטור מתמודד עם סוגיות מרכזיות כמו תלויות נתונים, קוהרנטיות מטמונים, וניהול משאבים משותפים.

2. מבנה כללי של המערכת

2.1. ארכיטקטורת המערכת

הארכיטקטורה המרכזית של המערכת מבוססת על ארבע ליבות עיבוד זהות, כאשר כל ליבה מהווה יחידת עיבוד עצמאית. כל ליבה מכילה זיכרון הוראות פרטי בגודל 1024 שורות, המאפשר אחסון והרצה של תוכניות באופן עצמאי. בנוסף, לכל ליבה מטמון נתונים פרטי בגודל 256 מילים, המשפר את ביצועי הגישה לזיכרון. הליבות מחוברות לזיכרון ראשי משותף באמצעות באס המיישם את פרוטוקול הקוהרנטיות MESI. בכל ליבה מיושמת צנרת מתקדמת בת חמישה שלבים: שליפה, פענוח, ביצוע, זיכרון, וכתיבה חזרה. הצנרת תוכננה ללא מנגנוני מעקפים, בחירה המפשטת את המימוש אך מחייבת טיפול קפדני בתלויות נתונים באמצעות השהיות. כל ליבה כוללת 16 רגיסטרים כלליים ברוחב 32 סיביות, כאשר חלקם מיועדים למטרות מיוחדות.

2.2. מערכת הזיכרון

מערכת הזיכרון בסימולטור מיישמת היררכיית זיכרון מורכבת הכוללת שלוש רמות, המיועדות להבטיח איזון בין ביצועים ליעילות תעבורת זיכרון. ברמה הראשונה, לכל ליבה יש זיכרון הוראות פרטי (Instruction Memory) המאחסן את התוכנית המיועדת לביצוע. זיכרון זה נפרד מזיכרון הנתונים ומבטיח שכל ליבה תוכל לגשת במהירות להוראות שהיא מבצעת, ללא תלות בפעולות אחרות במערכת.

ברמה השנייה של היררכיית הזיכרון, לכל ליבה יש מטמון נתונים פרטי המיושם כמטמון במיפוי ישיר. המטמון מחולק לבלוקים בגודל 4 מילים, ומנוהל באמצעות שני מבני נתונים עיקריים: DSRAM ו-TSRAM. ה-DSRAM מאחסן את התוכן של בלוקי הנתונים, בעוד שה-TSRAM מאחסן את התגיות (Tags) של הבלוקים ואת מצבי הקוהרנטיות שלהם בהתאם לפרוטוקול MESI. חלוקה זו מאפשרת גישה מהירה לנתונים המאוחסנים במטמון ומייעלת את השימוש בזיכרון.

מדיניות המטמון כוללת שימוש במנגנוני Write-Back ו-Write-Allocate. מנגנון Write-Back מבטיח שכתיבה לנתונים תתבצע רק במטמון, כאשר העדכון לזיכרון הראשי מתבצע רק כאשר הבלוק יוצא מהמטמון. מדיניות Write-Allocate, לעומת זאת,

מאפשרת הבאת נתונים לזיכרון המטמון גם במקרה של החטאה (Cache Miss) בפקודת כתיבה. שתי המדיניות הללו מפחיתות את תעבורת הבאס ומשפרות את יעילות התקשורת במערכת.

פרוטוקול MESI (Modified, Exclusive, Shared, Invalid) מנוהל ברמת המטמון ומשמש לשמירה על עקביות נתונים בין הליבות. כל בלוק במטמון יכול להימצא באחד מארבעה מצבים:

- Modified: הבלוק שונה במטמון הנוכחי ולא תואם לזיכרון הראשי. הוא ייכתב לזיכרון הראשי רק במקרה של פינוי בלוק מהמטמון.
- Exclusive: הבלוק נמצא במטמון הנוכחי בלבד והוא תואם לזיכרון הראשי. מצב זה מאפשר גישה מהירה ללא עדכון זיכרון.
- Shared: הבלוק משותף בין מספר מטמונים, ותואם לזיכרון הראשי. מצב זה מתרחש כאשר כמה ליבות משתמשות באותו נתון בו-זמנית.
- Invalid: הבלוק אינו תקף ואינו מתאים לשימוש. מצב זה מתרחש כאשר יש עדכון בבלוק על ידי ליבה אחרת או לאחר פינוי בלוק.

הטיפול בנתוני המטמון והמעבר בין המצבים מתבצע על ידי פונקציות ייעודיות בקוד. לדוגמה, פונקציית read_cache אחראית על קריאה לבלוק מהמטמון ובדיקה אם הנתון זמין (Cache Hit) או אם נדרשת פנייה לזיכרון הראשי במקרה של Cache Miss. במצב של החטאה, פונקציית update_state אחראית לעדכן את מצב הבלוק בהתאם לדרישות פרוטוקול MESI. במקביל, פונקציית write_cache מטפלת בכתיבה לבלוק במטמון ובמעבר למצבים Modified או Exclusive במידת הצורך.

כדי לנהל את התקשורת בין המטמונים והזיכרון הראשי, נעשה שימוש במנגנוני BusRd ו-BusRdX דרך הבאס. לדוגמה, במקרה של Cache Miss, הליבה שולחת בקשת BusRd לבאס, כדי לקבל את הנתונים מהזיכרון הראשי או ממטמון של ליבה אחרת. במצב של כתיבה לנתונים במצב Shared או Invalid, הליבה שולחת בקשת BusRdX, שמעדכנת את מצבי הקוהרנטיות בליבות האחרות ומאפשרת לליבה הנוכחית גישה בלעדית לנתון.

2.3. מערכת התקשורת

מערכת התקשורת במעבד מבוססת על באס משותף המהווה את ערוץ התקשורת המרכזי בין הליבות והזיכרון הראשי. הבאס תוכנן לתמוך בשלושה סוגי טרנזקציות בסיסיות: קריאת נתונים (BusRd), קריאה לצורך כתיבה (BusRdX), וכתיבה חזרה לזיכרון (Flush). מערכת זו מנוהלת על ידי שלושה רכיבים עיקריים שעובדים בתיאום מלא.

מנהל הבאס (**BusManager**) מהווה את עיקר מערכת התקשורת. הוא אחראי על הקצאת הבאס לליבות השונות באמצעות מנגנון ארביטרציה מסוג Round-Robin, המבטיח חלוקה הוגנת של זמן הבאס בין הליבות. מנגנון זה מונע מצבים בהם ליבה מסוימת נחסמת מגישה לבאס לאורך זמן. כאשר מספר ליבות מבקשות גישה לבאס באותו מחזור שעון, הליבה בעלת העדיפות הגבוהה ביותר מקבלת גישה, והעדיפות שלה יורדת בתור למחזור הבא, כך ששאר הליבות מקבלות הזדמנות הוגנת לגשת לבאס.

בכל ליבה פועלת יחידת בקשות (**BusRequestor**), האחראית על הממשק של הליבה מול הבאס. יחידה זו אחראית על הכנת בקשות גישה, שליחתן למנהל הבאס, והתמודדות עם תשובות שמתקבלות בחזרה. תפקידי היחידה כוללים ניהול של כל המידע הנדרש לביצוע טרנזקציה, כגון סוג הפעולה (קריאה, כתיבה או ניקוי נתונים), כתובת המטרה בזיכרון, והנתונים במקרה של כתיבה. יחידת הבקשות מתזמנת את הפעולות כך שליבה לא תשלח בקשה חדשה לפני שהבקשה הקודמת שלה הסתיימה במלואה.

יחידת ההאזנה (**BusSnooper**) היא רכיב קריטי במערכת התקשורת, והיא פועלת בכל ליבה כדי לעקוב אחר הפעולות שמתרחשות בבאס ולעדכן את מצבי המטמון בהתאם. לדוגמה, כאשר פקודת BusRd מתבצעת בבאס, יחידת ההאזנה בודקת אם הנתון המבוקש נמצא במטמון המקומי. במקרה של פקודת BusRdX, היחידה תבטל את תקפותו של בלוק מטמון מקומי אם הוא אינו בלעדי יותר. בנוסף, במצב של Flush, יחידת ההאזנה מתעדכנת שהנתונים נכתבו מהמטמון לזיכרון הראשי, ומבטיחה שמצב הקוהרנטיות של הבלוקים יתעדכן בהתאם.

שלושת סוגי הטרנזקציות בבאס מאפשרים ניהול יעיל של הזיכרון המשותף. פקודת BusRd מופעלת כאשר ליבה צריכה לקרוא נתון שאינו נמצא במטמון המקומי שלה (Cache Miss), ובמקרה זה הבאס מספק את הנתונים מהמקום המתאים – זיכרון ראשי או מטמון של ליבה אחרת. פקודת BusRdX מופעלת כאשר ליבה מבקשת לשנות נתון שנמצא במצב Shared או Invalid, ובכך נדרש להעניק לה בלעדיות על הנתון במצב Modified. פקודת Flush מופעלת כאשר יש לכתוב את הנתונים מהמטמון לזיכרון הראשי, לדוגמה במצב בו המטמון מלא ויש לפנות מקום לנתונים חדשים.

הרכיבים BusManager, BusRequestor, וBusSnooper פועלים בתיאום מלא כדי להבטיח את תפקוד המערכת. מנהל הבאס מבצע חלוקת עדיפויות הוגנת, יחידות הבקשה מוודאות שליבות מנהלות את פעולותיהן בצורה מסודרת ויעילה, ויחידות ההאזנה דואגות לשמירה על עקביות הנתונים במטמונים לפי פרוטוקול MESI. מערכת זו מבטיחה תקשורת אפקטיבית, ניהול משאבים תקין, ושמירה על עקביות בין הליבות, תוך תמיכה בהרצה מקבילית ויעילה של יישומים.

2.4. מימוש הצנרת

צנרת הביצוע בכל ליבה מורכבת מחמישה שלבים המתוזמנים במדויק. בשלב השליפה (**Fetch**), המערכת קוראת הוראה מזיכרון ההוראות הפרטי לפי ערך מונה התוכנית. שלב זה אחראי גם על זיהוי ראשוני של הוראות HALT ועדכון מונה התוכנית בהתאם להוראות הנוכחיות.

שלב הפענוח (**Decode**) מבצע את הפירוק המלא של ההוראה למרכיביה ומזהה תלויות נתונים אפשריות. במקרה של זיהוי תלות, השלב מפעיל מנגנון השהייה המעכב את התקדמות ההוראה עד לפתרון התלות. שלב זה אחראי גם על טיפול בהוראות קפיצה וחישוב כתובות היעד.

שלב הביצוע (**Execute**) מפעיל את יחידת החישוב האריתמטי-לוגית (ALU) בהתאם לסוג ההוראה. היחידה תומכת במגוון רחב של פעולות הכוללות חיבור, חיסור, כפל, פעולות לוגיות והזזות. עבור הוראות גישה לזיכרון, שלב זה מחשב את כתובת הגישה המדויקת.

בשלב הגישה לזיכרון (**Memory**) מתבצעת קריאה או כתיבה לנתונים בהתאם לסוג ההוראה. אם מדובר בהוראת קריאה שלב זה בודק האם הנתונים נמצאים במטמון (Cache Hit)וקורא אותם ישירות ממנו. במידה ויש החטאה במטמון מתבצעת פנייה לזיכרון הראשי דרך הבאס, והנתונים נטענים למטמון. עבור הוראת כתיבה השלב מעדכן את המטמון בהתאם למדיניות Write-Back תוך שמירה על עקביות הנתונים בין המטמון לזיכרון הראשי.

שלב הכתיבה חזרה (**Write-Back**) משלים את תהליך הביצוע על ידי עדכון הרגיסטרים בתוצאה הסופית. עבור הוראות חישוב, התוצאה מה ALU נכתבת לרגיסטר היעד (Rd) במקרה של הוראות קריאה, הנתונים שנשלפו מהמטמון או מהזיכרון הראשי נכתבים לרגיסטר היעד. שלב זה גם מתמודד עם מקרים מיוחדים, כמו אי עדכון רגיסטרים לדוגמה, רגיסטר R0 המשמש כקבוע אפס ומעדכן את המערכת להוראה הבאה.

2.5. הסימולטור

הסימולטור מהווה את הליבה המרכזית של הפרויקט ומאפשר לדמות את פעולתה של מערכת מרובת ליבות באופן מדויק ומפורט. מטרתו היא לספק תשתית לבחינה מעמיקה של תהליכים המתרחשים בארכיטקטורת המעבד, כולל ניהול צנרת הליבות, תקשורת בין רכיבים באמצעות באס, ניהול זיכרון, והבטחת עקביות באמצעות פרוטוקול MESI. הסימולטור בנוי כך שיאפשר ניתוח ביצועים, פתרון בעיות, ולמידה של אתגרים בתכנון מערכות מחשב מורכבות.

2.5.1. מטרות הסימולטור

הסימולטור ממלא מספר תפקידים מרכזיים. ראשית, הוא מתאם את פעולת הצנרת בכל אחת מארבע הליבות, כולל פתרון בעיות של תלות נתונים וניהול השהיות. שנית, הוא מטפל בניהול הזיכרון באמצעות סימולציה של קריאות וכתיבות במטמון המקומי ובזיכרון הראשי. בנוסף, הסימולטור מנהל את התקשורת בין הליבות, המטמונים והזיכרון המשותף באמצעות באס מרכזי. לבסוף, הסימולטור אוסף נתונים על ביצועי המערכת, כמו זמני שעון, שיעור פגיעות והחטאות במטמון, וניצול יעיל של הבאס.

2.5.2. מבנה הסימולטור

הסימולטור בנוי ממספר רכיבים מרכזיים הפועלים בתיאום. הרכיב הראשון הוא הלולאה המרכזית שמנהלת את מחזורי השעון. בכל מחזור שעון, מתבצע תזמון של כל פעולות הליבות, ניהול תעבורת הבאס, ועדכון מצבי המטמונים והזיכרון. בנוסף, לכל ליבה יש מנגנון עצמאי המאפשר לה לעבור בין חמשת שלבי הצנרת שלה. רכיב נוסף הוא ניהול הבאס, הכולל חלוקה הוגנת של משאבי הבאס בין הליבות באמצעות מנגנון Round-Robin. לבסוף, קיימים רכיבים לניהול המטמונים והזיכרון המשותף, המטפלים בכל פעולת גישה למידע.

2.5.3. תהליך ריצת הסימולציה

תהליך הסימולציה מחולק לשלושה שלבים עיקריים: אתחול, ריצה ותיעוד סופי. בשלב האתחול, כל רכיבי המערכת מאותחלים לערכי ברירת המחדל שלהם. פונקציות כמו initialize_simulator_objects ו reset_main_memory-משמשות להגדרת מצב הליבות, המטמונים והזיכרון הראשי. בשלב הריצה, הסימולטור מבצע מחזורי שעון, שבהם כל ליבה מבצעת את שלבי הצנרת שלה. רכיב הבאס מנהל את תעבורת המידע בין המטמונים לזיכרון הראשי, ומערכת ההאזנה (BusSnooper) מעדכנת את מצבי המטמונים בהתאם לפרוטוקול MESI. השלב האחרון הוא שלב התיעוד, שבו נשמרים קבצי פלט המפרטים את מצב המערכת, כמו מצב הרגיסטרים, המטמונים והזיכרון הראשי, יחד עם סטטיסטיקות ביצועים.

2.5.4. פונקציות עיקריות בסימולטור

הסימולטור עושה שימוש במגוון פונקציות ייעודיות לניהול הרכיבים השונים. לדוגמה, פונקציית do_fetch_operation מטפלת בשליפת הוראות מזיכרון ההוראות הפרטי, בעוד ש do_decode_operation אחראית על פענוח ההוראות וזיהוי תלות בנתונים. פעולות גישה למטמון, כמו handleCacheHit ו handleCacheMiss מטפלות בקריאות וכתיבות במטמון ובפנייה לזיכרון הראשי במקרה של החטאה. פונקציית write_next_cycle_of_bus מתזמנת את פעולות הבאס ומנהלת את העדיפויות בין הליבות.

2.5.5. מעקב אחרי ביצועים

הסימולטור אוסף נתונים סטטיסטיים על ביצועי המערכת לאורך כל הסימולציה. בין הנתונים הנאספים ניתן למצוא את שיעור הפגיעות וההחטאות במטמון, זמני ההמתנה של הליבות בבאס, ומספר מחזורי השעון הנדרשים לביצוע כל הוראה. נתונים אלה נשמרים בקבצי פלט המאפשרים ניתוח מעמיק של יעילות המערכת וזיהוי צווארי בקבוק.

2.5.6. אתגרים ופתרונות

- **ניהול השהיות בצנרת (Stalls)**

אחד האתגרים המרכזיים בפיתוח הסימולטור היה ניהול תלויות נתונים והשהיות בצנרת בת חמישה שלבים. כאשר הוראה מסוימת תלויה בתוצאת חישוב של הוראה קודמת (לדוגמה, קריאה מהרגיסטר שמחושב רק בשלב ה Write-Back) יש לעכב את ההוראה הנוכחית עד שהתוצאה תהיה זמינה. הבעיה הזו טופלה באמצעות מנגנון השהייה מובנה בכל שלב בצנרת. פונקציות כמו do_decode_operation בודקות את מצב הרגיסטרים ועדכניותם לפני המשך הפענוח. במקרים של תלות בנתונים, הליבה משאירה את ההוראה הנוכחית בשלבה הנוכחי וממתינה עד לסיום ההוראה הקודמת, תוך שימוש במונה מחזורי שעון לניהול ההשהיה.

- **קוהרנטיות במטמונים (Cache Coherence)**

שמירה על עקביות הנתונים בין המטמונים של הליבות הייתה אתגר מהותי, במיוחד במערכת מרובת ליבות. כל שינוי בנתון במטמון של ליבה אחת יכול להשפיע על המטמונים של ליבות אחרות. האתגר הזה טופל באמצעות פרוטוקול MESI שמיושם בקבצים כמו bus_snooper.c ו cache_block.c. רכיב ה BusSnooper מאזין לפעולות בבאס (כמו BusRd וBusRdX) ומעדכן את מצבי הבלוקים במטמון. לדוגמה, כאשר ליבה אחת שולחת בקשת BusRdX על בלוק משותף, ה BusSnooper בליבות האחרות משנה את מצב הבלוק ל Invalid- כך מובטחת עקביות בין המטמונים תוך מינימום פגיעה בביצועים.

- **עומס על הבאס וניהול עדיפויות**

במערכת מרובת ליבות, הבאס מהווה משאב משותף וקריטי, ולעיתים מספר ליבות מתחרות על גישה אליו באותו זמן. הדבר יצר צוואר בקבוק שעלול לעכב את ביצוע ההוראות. האתגר הזה נפתר באמצעות מנגנון ארביטרציה מסוג Round-Robin, המיושם בקובץ bus_manager.c. מנגנון זה מבטיח חלוקה הוגנת של זמני הבאס בין הליבות, כך שליבה אחת לא "תנעל" את הבאס על חשבון אחרות. פונקציה כמו finish_bus_enlisting מתעדפת ליבות בצורה סיבובית, בעוד שהפונקציה advance_bus_to_next_cycle מבצעת רוטציה בעדיפויות לאחר כל מחזור.

- **טיפול במצבי החטאה במטמון (Cache Miss)**

כאשר ליבה מבצעת גישה לנתון שאינו נמצא במטמון (Cache Miss), היה צורך לפתח מנגנון שיביא את הנתון מהזיכרון הראשי או ממטמון של ליבה אחרת בצורה יעילה. האתגר טופל באמצעות שילוב של פעולות BusRd ו BusRdX שמבוצעות על ידי פונקציות כמו handleCacheMiss בקובץ memory_stage.c. במצב של החטאה, הפונקציה מבצעת קריאה מהזיכרון הראשי ומעדכנת את המטמון המקומי לפי מדיניות Write-Allocate. אם הנתון נמצא במטמון של ליבה אחרת, פרוטוקול MESI מבטיח שהנתון יישלח מהמטמון הרלוונטי ולא יילקח מהזיכרון הראשי.

- **כתיבה לזיכרון הראשי (Flush)**

במקרה של פינוי בלוק מהמטמון, היה צורך להבטיח שכתיבה לזיכרון הראשי תתבצע באופן תקין תוך שמירה על ביצועים. האתגר הזה טופל באמצעות מדיניות Write-Back. כאשר בלוק במטמון נמצא במצב Modified והוא מפונה, הנתון נכתב לזיכרון הראשי באמצעות פעולת Flush. בקובץ bus_snooper.c פונקציות כמו do_snoop_operation עוקבות אחרי פעולות Flush ומוודאות שהנתון אכן נכתב לזיכרון הראשי. כך ניתן להקטין את תעבורת הבאס ולשפר את הביצועים הכוללים.

- **ניטור ביצועים וזיהוי צווארי בקבוק**

אחד האתגרים היה לעקוב אחר ביצועי המערכת ולזהות צווארי בקבוק אפשריים, כמו שיעור החטאות במטמון או ניצול נמוך של הבאס. הדבר טופל על ידי איסוף סטטיסטיקות באמצעות פונקציות ב simulator.c כמו רישום כמות הפגיעות והחטאות

במטמון וזמני ההמתנה בבאס. הנתונים נשמרים בקבצי פלט ומשמשים לניתוח מעמיק של תפקוד המערכת. באמצעות המידע הזה ניתן לבצע אופטימיזציות, כמו שינוי גודל המטמון או התאמת מנגנון ניהול הבאס.

3. מבנה הקבצים והקוד

הפרויקט מחולק למספר קבצי קוד בשפת C וקבצי Header, המייצגים רכיבים שונים במערכת הסימולציה. הקבצים עובדים יחד בסינכרון כדי לממש את הצנרת, המטמון, הבאס והזיכרון הראשי. המבנה המודולרי מאפשר תחזוקה קלה והרחבה של הפרויקט.

3.1. קבצי קוד עיקריים

3.1.1. simulator.c

simulator.c הוא הקובץ המרכזי בפרויקט, המהווה את הליבה של מערכת הסימולציה כולה. קובץ זה אחראי על ניהול מחזורי השעון של הסימולציה, תיאום פעולת הליבות, ניהול הבאס, עדכון מצב הזיכרון הראשי, ולבסוף כתיבת פלטים המסכמים את ביצועי המערכת. הקובץ מבצע את האתחול הראשוני של כל רכיבי המערכת, מפעיל את הסימולציה במחזוריות, ומספק כלי לאיסוף וניתוח נתונים סטטיסטיים.

הפונקציה **initialize_simulator_objects()** אחראית לאתחול כל רכיבי המערכת. במהלך תהליך זה נוצרות ליבות העיבוד, מוקצים מבני הזיכרון הראשי, ומוגדר מנהל הבאס שמסנכרן בין הרכיבים. בנוסף, הפונקציה טוענת קבצי קלט חשובים, כגון imem.txt המייצג את זיכרון ההוראות של הליבות, ו memin.txt המשמש כזיכרון הראשי. לאחר טעינת הקבצים, מתבצעת קריאה לפונקציות אתחול נוספות כמו configure_core עבור כל ליבה בנפרד, ו configure_bus_manager לאתחול רכיב הבאס.

הלולאה הראשית של הסימולציה ממומשת בפונקציה run_simulation(). בכל מחזור שעון, מופעלים חמשת שלבי הצנרת (Fetch, Decode, Execute, Memory, Write Back) עבור כל אחת מארבע הליבות. הלולאה ממשיכה לפעול עד שכל הליבות מגיעות להוראת HALT,שמסמנת את סיום התוכנית שלהן. במהלך הריצה, הסימולטור מנטר את מצב הליבות ומבצע תיאום ביניהן, כולל סנכרון גישות לבאס במקרים של תחרות בין הליבות.

לאחר סיום הסימולציה, הפונקציה **write_simulation_outputs()** אחראית לכתיבת קבצי הפלט המסכמים את ריצת הסימולציה. הקבצים כוללים את regout.txt שמתעד את מצב הרגיסטרים של כל ליבה בסיום הריצה, ו memout.txt המציג את מצב הזיכרון הראשי. בנוסף bus_trace.txt מתעד את כל הטרנזקציות שבוצעו בבאס במהלך הריצה, ו core_traceX.txt (כאשר X מציין את מספר הליבה) מתעד את ההוראות שבוצעו בכל ליבה ומעקב אחרי התקדמות הצנרת.

במקביל, מתבצע איסוף נתונים סטטיסטיים על ביצועי המערכת באמצעות הפונקציה **collect_statistics()**. פונקציה זו מחשבת את מספר הפגיעות (Hits) וההחטאות (Misses) במטמון עבור כל ליבה, ומנטרת את זמני ההמתנה של הליבות לבאס, תוך ניתוח השפעתם על ביצועי המערכת הכוללים. הנתונים שנאספים נכתבים לקבצי statsX.txt

ומאפשרים ניתוח מעמיק של הביצועים, זיהוי צווארי בקבוק, וביצוע אופטימיזציות במידת הצורך.

3.1.2. core.c

core.c מממש את הליבה הבודדת של המעבד ומרכז את כל הלוגיקה הקשורה לביצוע הוראות. הקובץ אחראי על ניהול חמשת שלבי הצנרת (Pipeline) של הליבה, כולל שליפת הוראות, פענוח, ביצוע חישובים, גישה לזיכרון, וכתיבה חזרה לרגיסטרים. בנוסף, הקובץ מנהל את האינטראקציה של הליבה עם מערכת הזיכרון והבאס, כולל ניהול השהיות (Stalls) במצבים של תלויות נתונים או עיכובים בגישה לזיכרון.

הפונקציה **configure_core()** אחראית לאתחול הליבה. בתהליך זה מאופסות כל ההגדרות של הליבה, כולל אתחול של 32 רגיסטרים (כאשר R0 משמש כרגיסטר קבוע עם ערך אפס). כמו כן, נטען זיכרון ההוראות (instruction_memory) מקובץ הקלט המתאים לכל ליבה. מעבר לכך, הפונקציה מאתחלת את חמשת שלבי הצנרת באמצעות קריאות לפונקציות כמו configure_fetch_stage, configure_decode_stage וכן הלאה, ומוודאת שכל שלב בצנרת מוכן לתחילת הסימולציה.

לאחר אתחול הליבה, הפונקציה **advance_core()** אחראית לקידום הליבה למחזור השעון הבא. בשלב זה מתבצעת העברת נתונים בין שלבי הצנרת באמצעות מנגנוני Flip-Flops, המבטיחים שהמידע מהשלב הקודם יעבור בצורה נכונה לשלב הבא. בנוסף, מנוהלות השהיות (Stalls) במקרים שבהם יש תלויות בנתונים או עיכובים בגישה לבאס. לדוגמה, אם הוראה בשלב הפענוח תלויה בתוצאה של הוראה שנמצאת בשלבי ביצוע מאוחרים יותר, הליבה תושהה עד שהנתונים יהיו זמינים.

בשלב **do_fetch_operation()** הליבה שולפת את ההוראה הבאה לביצוע מזיכרון ההוראות (instruction_memory). במהלך שלב זה, מונה התוכנית (PC) מתעדכן בהתאם להתקדמות הביצוע, ובמקרה של זיהוי הוראת HALT, הליבה מסמנת את סיום הביצוע שלה.

הפונקציה **do_decode_operation()** אחראית על פענוח ההוראה. בשלב זה, הליבה מזהה את סוג הפעולה שעליה לבצע (לדוגמה, חיבור, קריאה מזיכרון, או קפיצה מותנית), ובודקת אם קיימות תלויות בנתונים (Data Hazards). אם זוהתה תלות בנתונים, לדוגמה כאשר הוראה תלויה בתוצאה של חישוב שטרם הסתיים, הליבה תושהה עד שהתלות תיפתר.

בשלב **do_execute_operation()** מתבצע החישוב הלוגי או האריתמטי באמצעות יחידת החישוב (ALU). הליבה תומכת במגוון פעולות: פעולות אריתמטיות, פעולות לוגיות ופעולות הזזה.

הפונקציה **do_memory_operation()** אחראית על ביצוע פעולות קריאה (LW) או כתיבה (SW) לנתונים במטמון או בזיכרון הראשי. הליבה בודקת אם הנתון נמצא במטמון (Cache Hit) או שיש לבצע גישה לזיכרון הראשי במקרה של החטאה (Cache Miss). במצבים של Cache Miss נשלחת בקשת גישה לבאס, והליבה ממתינה לאישור לבצע את הפעולה.

לבסוף, הפונקציה **do_writeback_operation()** מבצעת את שלב הכתיבה חזרה לרגיסטרים. לדוגמה, אם התבצעה הוראת חיבור (ADD) התוצאה תישמר ברגיסטר היעד (RD). במקרים של הוראות קריאה מזיכרון, הנתונים שהובאו מהמטמון או

מהזיכרון הראשי יכתבו לרגיסטר המתאים. שלב זה מסמן את סיום מחזור ההוראה בצנרת הליבה.

3.1.3. cache_block.c

cache_block.c מממש את מערכת המטמון (Cache) של כל ליבה במעבד, ומרכז את כל הלוגיקה הקשורה לניהול קריאות וכתיבות לנתונים. הקובץ מטפל בזיהוי פגיעות (Cache Hits) והחטאות (Cache Misses) ומיישם את פרוטוקול MESI לשמירה על עקביות הנתונים בין הליבות במערכת מרובת ליבות. המטמון בנוי בשיטת מיפוי ישיר (Direct Mapping) וכל בלוק במטמון מנוהל במבני נתונים ייעודיים.

הפונקציה **reset_cache()** אחראית על אתחול המטמון לערכים התחלתיים בתחילת הסימולציה. המטמון מנוהל באמצעות שני מבנים עיקריים:

- **DSRAM**- מאחסן את הנתונים עצמם.
- **TSRAM**- מאחסן את תגיות הכתובות (Tags) ואת מצבי הקוהרנטיות (MESI) של כל בלוק.

בעת האתחול, כל הבלוקים במטמון מוגדרים במצב Invalid וכל הערכים מנוקים כדי להבטיח שהמטמון נקי ממצבים קודמים.

הפונקציה **read_cache(address)** מבצעת קריאה לנתון ממטמון הליבה. כאשר מתבצעת קריאה, המטמון בודק אם הנתון כבר נמצא בו:

- אם הנתון נמצא במטמון (Cache Hit) הוא נשלף ישירות מ DSRAM והביצוע ממשיך ללא עיכוב.
- אם הנתון אינו נמצא במטמון (Cache Miss) הליבה שולחת בקשת BusRd לבאס כדי לקרוא את הנתון מהזיכרון הראשי או ממטמון של ליבה אחרת. במצב כזה, הנתון ייטען למטמון בהתאם למדיניות Write-Allocate.

כתיבה לנתונים במטמון מתבצעת באמצעות הפונקציה **write_cache(address, data)** שפועלת בהתאם למדיניות Write-Back :

- אם הבלוק שבו מתבצעת הכתיבה נמצא במצב Modified או Exclusive הנתון נכתב ישירות למטמון מבלי לעדכן את הזיכרון הראשי באופן מיידי.
- אם הבלוק נמצא במצב Shared הליבה חייבת להשיג גישה בלעדית לבלוק לפני כתיבה. במקרה כזה, נשלחת בקשת BusRdX לבאס כדי לעדכן את מצבי הקוהרנטיות בליבות האחרות, ולאחר מכן מתבצעת הכתיבה.

הפונקציה **update_state()** מטפלת בעדכון מצב הבלוק במטמון בהתאם לפרוטוקול MESI. כל בלוק במטמון יכול להיות באחד מהמצבים הבאים:

- **Modified:** הבלוק שונה במטמון ואינו תואם לזיכרון הראשי. הנתון ייכתב לזיכרון הראשי רק כאשר הבלוק יפונה מהמטמון.
- **Exclusive:** הבלוק נמצא רק במטמון הנוכחי ותואם לזיכרון הראשי. במצב זה ניתן לכתוב לבלוק מבלי לשלוח בקשה לבאס.

- **Shared** : הבלוק נמצא במספר מטמונים ותואם לזיכרון הראשי. במצב זה נדרשת בקשת BusRdX לפני ביצוע כתיבה.
- **Invalid** : הבלוק אינו תקף לשימוש, ויש לטעון אותו מחדש מהזיכרון הראשי או ממטמון אחר.

באמצעות מימוש זה, מערכת המטמון מצליחה לייעל את הגישה לנתונים ולשמור על עקביות במערכת מרובת ליבות. השילוב בין ניהול פגיעות והחטאות, ושמירה על קוהרנטיות הנתונים בעזרת פרוטוקול MESI מאפשר ביצועים גבוהים תוך שמירה על תקינות המידע בין הליבות.

3.1.4. bus_manager.c

bus_manager.c אחראי על ניהול הבאס במערכת, שהוא ערוץ התקשורת המרכזי בין הליבות לזיכרון הראשי. קובץ זה מנהל את חלוקת הגישות לבאס בצורה הוגנת ויעילה, מונע קונפליקטים בין הליבות, ומבטיח שהגישה לזיכרון ולמטמונים תתבצע באופן מסונכרן. הבאס מנוהל כך שיאפשר עבודה תקינה במערכת מרובת ליבות, תוך שמירה על עקביות בזיכרון באמצעות פרוטוקול MESI.

הפונקציה **configure_bus_manager()** מאתחלת את רכיב מנהל הבאס ומבצעת את החיבורים הדרושים לליבות ולזיכרון הראשי. במהלך תהליך זה, מוגדרים תורי הבקשות של הליבות לבאס, ומוקצים מבני נתונים לניהול טרנזקציות עתידיות. כל ליבה מקבלת מזהה ייחודי המאפשר לה לתקשר עם הבאס בצורה מסודרת, וכן להאזין לפעולות שמבוצעות בו.

פונקציית **manage_bus_access()** אחראית לניהול חלוקת הגישות לבאס בין הליבות. היא פועלת באמצעות מנגנון Round-Robin שמבטיח חלוקה הוגנת של זמן הבאס בין הליבות השונות. כאשר מספר ליבות מבקשות גישה לבאס באותו מחזור שעון, הליבה בעלת העדיפות הגבוהה ביותר תקבל גישה ראשונה. העדיפות מתעדכנת בכל מחזור, כך שלכל ליבה תהיה הזדמנות הוגנת לבצע את פעולתה ללא חסימה ממושכת.

הפונקציה **write_next_cycle_of_bus()** מעדכנת את מצב הבאס בהתאם לבקשות שהתקבלו מהליבות. היא אחראית לתעד את כל הטרנזקציות שבוצעו בבאס, כולל פעולות קריאה (BusRd), כתיבה בלעדית (BusRdX) וכתיבה חזרה לזיכרון (Flush). כל טרנזקציה מתועדת בקובץ bus_trace.txt המאפשר מעקב מדויק אחרי פעולות הבאס לאורך הסימולציה.

לבסוף, פונקציית **process_bus_transactions()** מטפלת בעיבוד הטרנזקציות בבאס. היא מבצעת את הבדיקות הדרושות עבור כל בקשה, מטפלת בבקשות קריאה וכתיבה, ומעדכנת את מצב הזיכרון הראשי והמטמונים בהתאם. לדוגמה, כאשר מתקבלת בקשת BusRd, הבאס מספק את הנתון הנדרש מהזיכרון הראשי או ממטמון של ליבה אחרת. כאשר מתקבלת בקשת Flush, הנתון מהמקטע הרלוונטי במטמון נכתב חזרה לזיכרון הראשי. באמצעות תהליך זה, הבאס מצליח לנהל את התקשורת בין הליבות והזיכרון בצורה חלקה ויעילה, תוך שמירה על עקביות הנתונים במערכת.

3.1.5. bus_snooper.c

bus_snooper.c מממש את מנגנון ההאזנה לבאס (Bus Snooping) שהוא חלק קריטי בשמירה על עקביות הנתונים במערכת מרובת ליבות. מנגנון זה מאפשר לכל ליבה לעקוב אחרי פעולות שמבוצעות בבאס ולוודא שמצבי המטמון שלה מעודכנים בהתאם לפרוטוקול MESI. בזכות ההאזנה, כל ליבה יכולה לזהות שינויים בנתונים המשותפים לליבות אחרות ולמנוע מצב שבו ליבות שונות עובדות עם נתונים לא עקביים.

הפונקציה **do_snoop_operation()** אחראית על האזנה מתמדת לפעולות בבאס. בכל מחזור שעון, הליבה בודקת את סוג הטרנזקציות המבוצעות בבאס ומתאימה את מצבי המטמון שלה בהתאם. לדוגמה, כאשר ליבה אחרת מבצעת פקודת BusRdX (בקשת קריאה עם כוונה לכתיבה), הליבה מאזינה ומעדכנת את מצב הבלוקים הרלוונטיים במטמון שלה למצב Invalid. עדכון זה מונע מליבה להשתמש בנתונים שאינם עדכניים, ומבטיח שכל גישה עתידית לבלוק זה תדרוש קריאה מחודשת מהזיכרון הראשי או מהמטמון של ליבה אחרת.

הפונקציה **handle_flush()** מטפלת בכתיבה של נתונים מזיכרון המטמון לזיכרון הראשי, תהליך הידוע כ Flush. כתיבה זו מתבצעת כאשר בלוק במטמון נמצא במצב Modified וצריך להתפנות, למשל במקרה של החטאה במטמון (Cache Miss) או כאשר המטמון מתמלא ויש צורך לפנות מקום לבלוקים חדשים. הפונקציה דואגת לשלוח את הנתון המעודכן מהמטמון לזיכרון הראשי לפני שהבלוק נמחק או משתנה, ובכך שומרת על עקביות הנתונים בין כל הליבות והזיכרון הראשי.

באמצעות מנגנון ההאזנה, כל ליבה שומרת על סנכרון מלא עם שאר הליבות במערכת, ומבטיחה שהנתונים במטמון תמיד יהיו עקביים עם שאר המערכת. זהו מרכיב חיוני במערכת מרובת ליבות, המאפשר למעבד לעבוד בצורה תקינה ויעילה תוך שמירה על תקינות הנתונים.

3.1.6. main_memory.c

main_memory.c אחראי על ניהול הזיכרון הראשי (Main Memory) של המערכת. הזיכרון הראשי הוא רכיב קריטי במערכת הסימולציה, מכיוון שהוא משמש כנקודת האחסון המרכזית לנתונים שאינם זמינים במטמונים של הליבות. כל גישה לזיכרון מתבצעת דרך הבאס, במיוחד במקרים של החטאות במטמון (Cache Miss) או בעת פינוי בלוקים במצב Modified.

הפונקציה **reset_main_memory()** אחראית על אתחול הזיכרון הראשי לערכים התחלתיים בתחילת הסימולציה. תהליך האתחול כולל ניקוי של כל התאים בזיכרון הראשי והגדרתם לערך אפס, כדי להבטיח שהמערכת תתחיל ממצב נקי. בנוסף, פונקציה זו טוענת את תוכן הקלט מקובץ memin.txt שמכיל את המידע הדרוש להתחלת הריצה, כולל נתונים ראשוניים שעליהם תתבצע הסימולציה.

הפונקציה **read_main_memory(address)** מבצעת קריאה של נתון מהזיכרון הראשי. קריאה זו מתבצעת בעיקר כאשר יש Cache Miss במטמון של אחת הליבות, כלומר כאשר הנתון המבוקש אינו קיים במטמון ויש צורך לקרוא אותו ישירות מהזיכרון הראשי. הפונקציה מקבלת כקלט את הכתובת הרצויה ומחזירה את הנתון המאוחסן בה. הקריאה מבוצעת לאחר בקשת BusRd דרך הבאס, ובהתאם לפרוטוקול MESI ייתכן שהנתון יגיע ממטמון של ליבה אחרת במקום מהזיכרון הראשי.

הפונקציה **write_main_memory(address, data)** מטפלת בכתיבה לנתון בזיכרון הראשי. פעולה זו מתבצעת כאשר בלוק במטמון נמצא במצב Modified וצריך להתפנות, לדוגמה כאשר מתבצעת החטאה במטמון או כשמטמון הליבה מלא ויש צורך לפנות מקום לנתונים חדשים. לפני פינוי הבלוק מהמטמון, הנתונים שנכתבו ושונו יועברו לזיכרון הראשי כדי לשמור על עקביות .תהליך זה מתבצע באמצעות בקשת Flush בבאס, שמבטיחה שהנתונים העדכניים יועברו בצורה מסודרת.

באמצעות פונקציות אלו main_memory.c, מבטיח את ניהול הזיכרון הראשי בצורה יעילה, תוך שמירה על קוהרנטיות נתונים במערכת מרובת ליבות. המימוש המדויק של קריאות וכתיבות בזיכרון הראשי מאפשר למערכת להתמודד עם מצבי Cache Miss בצורה אופטימלית ולשמור על ביצועים גבוהים לאורך הסימולציה.

3.2. קבצי Header

קבצי ה Header בפרויקט מהווים את הבסיס המבני למערכת הסימולציה כולה. כל קובץ מכיל הצהרות על מבני נתונים (structs) פונקציות, והגדרות קבועות המשמשות ליצירת אינטגרציה חלקה בין הרכיבים השונים במערכת. בעזרת חלוקת הקוד למודולים ברורים, ניתן לנהל, להרחיב, ולתחזק את המערכת בצורה מסודרת ויעילה. להלן פירוט מקיף ומסודר של כל קבצי ה Header בפרויקט.

3.2.1. bus_manager.h

קובץ bus_manager.h מגדיר את מבנה BusManager שהוא הרכיב המרכזי האחראי על ניהול ותיאום הגישות לערוץ התקשורת המרכזי במערכת – הבאס (Bus). תפקידו של מנהל הבאס הוא להבטיח גישה מסודרת ומבוקרת בין הליבות השונות לבין הזיכרון הראשי, תוך מניעת קונפליקטים וניהול תעדופים בין הבקשות השונות שמתקבלות מהליבות. הבאס מהווה צוואר בקבוק פוטנציאלי במערכות מרובות ליבות, ולכן ניהול יעיל שלו הוא קריטי לשמירה על ביצועים אופטימליים ועקביות נתונים.

מבנה ה BusManager כולל מספר שדות חשובים לניהול יעיל של הבאס. אחד השדות המרכזיים הוא **bus_status** שמתאר את מצב הבאס הנוכחי – האם הוא פנוי (BUS_FREE) או עסוק בביצוע טרנזקציה כלשהי, כמו קריאה (BUS_RD) או קריאה לצורך כתיבה (BUS_RDX). שדה נוסף **last_transaction_on_bus_cycle**, מתעד את מחזור השעון האחרון שבו בוצעה טרנזקציה על הבאס, מידע שחשוב למעקב אחרי תפקוד המערכת ולניהול תעדופים בין הליבות. מעבר לכך, מנהל הבאס מחזיק מערך של מבקשי באס (requestors) כאשר כל אובייקט במערך מייצג ליבה במערכת שיכולה להגיש בקשה לביצוע פעולה בבאס.

בין הפונקציות העיקריות שמוגדרות בקובץ ניתן למצוא את **configure_bus_manager()** שאחראית על אתחול רכיב מנהל הבאס. פונקציה זו יוצרת את הקשרים הנדרשים בין הבאס לליבות ולזיכרון הראשי, ומכינה את המערכת לתחילת הסימולציה. פונקציה נוספת **advance_bus_to_next_cycle()**, מקדמת את הבאס למחזור השעון הבא, תוך עדכון מצב הבאס בהתאם לטרנזקציות שבוצעו במחזור הקודם. פעולה זו מאפשרת לסימולציה להתקדם בזמן אמת ולנהל את זרימת

המידע בין הרכיבים השונים. לבסוף **enlist_to_bus()**, מוסיפה בקשה חדשה לגישה לבאס מתור הבקשות של הליבות, תוך ניהול נכון של סדר הבקשות ותעדופן.

באמצעות מבנה זה BusManager, מבטיח חלוקה הוגנת של משאבי התקשורת בין הליבות במערכת, תוך שמירה על עקביות הנתונים בזיכרון הראשי ובמטמונים המקומיים של כל ליבה. ניהול יעיל של הבאס מונע עיכובים מיותרים בביצוע הוראות ומאפשר למערכת לפעול בצורה סינכרונית ומדויקת, מה שמוביל לשיפור בביצועים הכוללים של המעבד הרב-ליבתי.

3.2.2. bus_requestor.h

קובץ bus_requestor.h מגדיר את מבנה BusRequestor שהוא הרכיב האחראי על ניהול בקשות הגישה לבאס מתוך כל ליבה במערכת. כל ליבה במעבד הרב-ליבתי משתמשת ברכיב זה על מנת לבצע קריאות וכתיבות לזיכרון הראשי או למטמון של ליבות אחרות. תפקידו של רכיב זה הוא לוודא שכל בקשה לבאס מתבצעת בצורה מסודרת, תוך שמירה על סדר העדיפויות ותיאום עם רכיבי ניהול נוספים כמו BusManager ו BusSnooper.

מבנה BusRequestor כולל מספר שדות חיוניים לניהול נכון של הבקשות. השדה המרכזי הוא operation המייצג את סוג הבקשה שהליבה שולחת לבאס – בין אם מדובר בקריאה (BusRd), קריאה לצורך כתיבה (BusRdX) או השהיה במקרים של עיכוב. שדה נוסף הוא address שמכיל את הכתובת בזיכרון שעליה מתבצעת הבקשה, ו data שבו מאוחסן הנתון שייכתב לזיכרון במקרה של בקשת כתיבה. מעבר לכך, ישנו שדה priority המאפשר לקבוע את סדר העדיפויות בגישה לבאס, ובכך מוודא חלוקה הוגנת של משאבי התקשורת במערכת. לבסוף, מצביע myCore מציין את הליבה שאליה שייך הרכיב, דבר המאפשר זיהוי וניהול נכון של הבקשות.

הפונקציה העיקרית המוגדרת בקובץ זה היא **configure_bus_requestor()** שאחראית על אתחול רכיב בקשת הבאס בליבה. פונקציה זו מגדירה את הערכים ההתחלתיים של המבנה, כולל סוגי הבקשות האפשריות, ומחברת את הרכיב לשאר המודולים במערכת כמו מנהל הבאס והזיכרון הראשי. באמצעות תהליך האתחול, כל ליבה מוכנה לשלוח בקשות לבאס ולנהל תקשורת עם רכיבים אחרים במערכת בצורה חלקה.

באמצעות מבנה זה, כל ליבה יכולה ליזום גישות לבאס ולבצע קריאות וכתיבות לזיכרון בצורה מבוקרת. הדבר מאפשר תיאום בין הליבות השונות, שמירה על עקביות הנתונים במערכת מרובת הליבות, ומניעת קונפליקטים בגישה לזיכרון המשותף. תכנון נכון של רכיב זה תורם ליעילות המערכת ומשפר את הביצועים הכוללים של המעבד.

3.2.3. bus_snooper.h

קובץ bus_snooper.h מגדיר את מבנה BusSnooper רכיב קריטי במערכת התקשורת של המעבד הרב-ליבתי. תפקידו המרכזי של רכיב זה הוא להאזין לטרנזקציות שמתרחשות בבאס ולעדכן את מצבי המטמון של הליבה בהתאם לפרוטוקול MESI (Modified, Exclusive, Shared, Invalid). מנגנון זה חיוני לשמירה על עקביות הנתונים במערכת מרובת ליבות, שבה כל ליבה מחזיקה עותקים מקומיים של נתונים

משותפים. בעזרת האזנה פעילה, ניתן למנוע מצבים שבהם ליבות שונות עובדות עם גרסאות לא מעודכנות של אותם נתונים.

מבנה BusSnooper כולל שדה יחיד בשם myCore שהוא מצביע לליבה שאליה שייך רכיב ההאזנה. מבנה פשוט זה מאפשר לכל ליבה לעקוב אחרי הטרנזקציות בבאס ולזהות פקודות קריטיות כמו BusRd (קריאת נתונים) ו BusRdX (קריאה לצורך כתיבה בלעדית). כאשר מתבצעת טרנזקציה רלוונטית בבאס, רכיב ההאזנה מזהה זאת ומעדכן את מצב הבלוקים במטמון המקומי של הליבה. לדוגמה, אם ליבה אחרת שולחת פקודת BusRdX על כתובת שנמצאת במטמון המקומי, הבלוק הרלוונטי במטמון ישונה למצב Invalid כך שהליבה לא תשתמש בנתונים שאינם עדכניים.

הקובץ כולל שתי פונקציות עיקריות:

- **configure_bus_snooper()** פונקציה זו מאתחלת את רכיב ההאזנה בליבה ומגדירה את מנגנון המעקב אחרי הבאס. היא מכינה את הרכיב לזיהוי טרנזקציות ומחברת אותו לשאר רכיבי המערכת הרלוונטיים, כמו מנהל הבאס והמטמון המקומי.
- **do_snoop_operation()** זוהי הפונקציה המרכזית שמבצעת את ההאזנה בזמן אמת. כאשר טרנזקציה מתבצעת בבאס, הפונקציה מזהה את סוג הפעולה ומבצעת את העדכונים הנדרשים במטמון הליבה בהתאם לכללי פרוטוקול MESI. כך, מצבי הבלוקים משתנים בהתאם לאירועים בבאס, ומובטחת עקביות הנתונים בכל הליבות במערכת.

באמצעות רכיב זה, המערכת שומרת על קוהרנטיות במטמון ,מה שמבטיח שכל ליבה תעבוד עם נתונים עדכניים ומדויקים. האזנה רציפה לטרנזקציות בבאס ומעקב אחרי פעולות ליבות אחרות הם מפתח לשמירה על תקינות הנתונים במערכת מרובת ליבות, תוך שמירה על ביצועים גבוהים ויעילות בסימולציה.

3.2.4. cache_block.h

קובץ cache_block.h מגדיר את מבנה מערכת המטמון (Cache) עבור כל ליבה במערכת המעבד הרב-ליבתי. המטמון מהווה רכיב קריטי לשיפור ביצועי המעבד בכך שהוא מספק גישה מהירה לנתונים בשימוש תדיר, ומפחית את הצורך בגישה לזיכרון הראשי האיטי יותר. הקובץ מתמקד בניהול הבלוקים במטמון, תוך שמירה על עקביות הנתונים בין הליבות באמצעות פרוטוקול MESI.

מבנה Cache מורכב משני רכיבי זיכרון עיקריים:

- dsram זיכרון הנתונים במטמון (Data SRAM) שבו מאוחסנים הערכים עצמם של הנתונים שנמצאים בשימוש הליבה. זהו זיכרון מהיר המאפשר גישה מיידית לנתונים.
- tsram זיכרון התגיות (Tag SRAM) שבו מאוחסנים תגיות הכתובות של הנתונים במטמון ומצבי הקוהרנטיות של הבלוקים בהתאם לפרוטוקול MESI. תגיות אלו מאפשרות לזהות האם הנתון במטמון תואם לנתון בזיכרון הראשי או שמא נדרש עדכון.

הקובץ כולל מספר פונקציות עיקריות המנהלות את פעולת המטמון:

- **reset_cache()** : פונקציה זו מאתחלת את מבנה המטמון לערכים התחלתיים. היא מנקה את כל התאים ב dsram וב- tsram ומגדירה את מצבי הבלוקים ל Invalid, מה שמבטיח שהמטמון מתחיל ממצב ריק ומוכן לטעינת נתונים חדשים.

- **read_cache(address)** : פונקציה זו מבצעת קריאה לבלוק מהמטמון. היא בודקת האם הכתובת המבוקשת קיימת במטמון (Cache Hit) או אם נדרש להביא את הנתון מהזיכרון הראשי (Cache Miss). במקרה של Cache Hit, הנתון נשלף ישירות מהמטמון ובמקרה של Cache Miss, תישלח בקשת קריאה לבאס כדי לקבל את הנתון מהזיכרון הראשי או ממטמון של ליבה אחרת.

- **write_cache(address, data)** : פונקציה זו כותבת נתון לבלוק במטמון בהתאם למדיניות Write-Back. במדיניות זו, הנתון נכתב תחילה למטמון, והעדכון לזיכרון הראשי מתבצע רק כאשר הבלוק מפונה מהמטמון. אם הבלוק במצב Modified או Exclusive - הכתיבה תתבצע ישירות במטמון, בעוד שבמקרה של מצב Shared - תישלח בקשת BusRdX לבאס לשם עדכון קוהרנטיות.

- **update_state()** : פונקציה זו מעדכנת את מצב הבלוק במטמון בהתאם לפרוטוקול MESI.היא משנה את מצב הבלוק ל Modified, Exclusive, Shared או Invalid בהתאם לטרנזקציות שמתרחשות בבאס ולפעולות שנעשו במטמון.

באמצעות מבנה ופונקציות אלו cache_block.h, מאפשר ניהול יעיל של המטמון בליבה, שיפור ביצועים בגישה לנתונים, ושמירה על עקביות בין ליבות שונות במערכת מרובת ליבות. הפרוטוקול והמבנה המוגדרים בקובץ זה תורמים להפחתת תעבורת הבאס ולייעול התקשורת במערכת.

3.2.5. constants.h

קובץ constants.h מרכז את כל ההגדרות הקבועות (constants) הדרושות לפעולת מערכת הסימולציה. קובץ זה משמש כנקודת ייחוס אחידה לכל פרמטרים החוזרים על עצמם במערכת, כמו גודל הזיכרון, מספר הליבות, פרוטוקול הקוהרנטיות במטמון, והגדרת אופקודים (Opcodes). באמצעות ריכוז ההגדרות במקום אחד, נשמרת עקביות בכל חלקי הקוד, והתחזוקה נעשית פשוטה יותר – כל שינוי בפרמטר אחד משתקף מיידית בכל המערכת.

קבועים עיקריים בקובץ זה כוללים את הגדרות מבנה המעבד והמערכת:

- **NUM_OF_CORES** מגדיר את מספר הליבות במערכת, שהוא 4 ליבות. כל ליבה פועלת כיחידת עיבוד עצמאית במסגרת הסימולציה, אך הן מתקשרות זו עם זו באמצעות באס משותף.

- **NUM_REGISTERS_PER_CORE** מגדיר את מספר הרגיסטרים הכלליים בכל ליבה. לכל ליבה מוקצים 16 רגיסטרים ברוחב 32 ביט, המשמשים לאחסון זמני של נתונים במהלך עיבוד ההוראות.

- **DATA_CACHE_WORD_DEPTH** מגדיר את גודל המטמון במילים. המטמון מכיל 256 מילים, ומאפשר גישה מהירה לנתונים תדירים, ובכך משפר את ביצועי הליבה.

- **MEMORY_DEPTH** מגדיר את גודל הזיכרון הראשי במערכת. הזיכרון הראשי כולל תאים המשמשים לאחסון קבוע של נתונים והוראות שאינן נמצאות במטמון.

בנוסף, constants.h מגדיר את מצבי הקוהרנטיות במטמון בהתאם לפרוטוקול MESI המבטיח עקביות נתונים במערכת מרובת ליבות. הפרוטוקול כולל ארבעה מצבים עיקריים:

- MODIFIED : הבלוק שונה במטמון ואינו תואם לזיכרון הראשי.
- EXCLUSIVE: הבלוק נמצא רק במטמון הנוכחי ותואם לזיכרון הראשי.
- SHARED : הבלוק נמצא במטמונים של מספר ליבות ותואם לזיכרון הראשי.
- INVALID: הבלוק אינו תקף ויש לקרוא אותו מחדש מהזיכרון הראשי או ממטמון אחר.

עוד חלק חשוב בקובץ זה הוא הגדרת האופקודים (Opcodes) המזהים את סוגי ההוראות הנתמכות במערכת. ההגדרות כוללות פקודות כמו:

- ADD, SUB, MUL- פקודות אריתמטיות לביצוע חיבור, חיסור וכפל.
- LW (Load Word) ו SW (Store Word)- פקודות לגישה לזיכרון, המאפשרות קריאה וכתיבה של נתונים מהזיכרון הראשי למטמון ולהפך.
- פקודות נוספות כוללות פעולות לוגיות והזזות ביטים.

באמצעות קובץ זה, כל החלקים במערכת משתמשים באותן הגדרות אחידות, מה שמבטיח עבודה עקבית ונכונה של הסימולטור כולו. יתרה מזאת, ריכוז ההגדרות בקובץ אחד מאפשר לבצע תחזוקה קלה ולבצע שינויים בפרמטרים קריטיים בצורה מהירה ופשוטה.

3.2.6. core.h

קובץ core.h מגדיר את מבנה ה Core המייצג ליבה בודדת במערכת הסימולציה של המעבד הרב-ליבתי. כל ליבה פועלת כיחידת עיבוד עצמאית, המממשת צנרת עיבוד בת חמישה שלבים ומבצעת הוראות באופן מקבילי עם ליבות אחרות. הקובץ כולל את כל הרכיבים הפנימיים של הליבה, החל ממונה התוכנית והרגיסטרים ועד למערכת המטמון ורכיבי התקשורת עם הבאס.

מבנה ה Core כולל את הרכיבים המרכזיים הבאים:

- **pc_register** : מונה התוכנית (Program Counter) המצביע על כתובת ההוראה הבאה לביצוע בזיכרון ההוראות. מונה זה מתקדם בהתאם להוראות בתוכנית וניתן לעדכנו בעת ביצוע קפיצות או קריאות פונקציה.
- **registers[32]**: מערך של 32 רגיסטרים ברוחב 32 ביט לכל ליבה. הרגיסטרים משמשים לאחסון זמני של נתונים במהלך עיבוד ההוראות, כולל ערכי ביניים של חישובים.
- **instruction_memory[1024]**: זיכרון ההוראות הפרטי של הליבה, המכיל עד 1024 הוראות שמועלות מהקלט בתחילת הסימולציה. זיכרון זה מופרד מזיכרון הנתונים ומאפשר לליבה לבצע קריאות הוראות בצורה עצמאית.

בנוסף, המבנה כולל את שלבי הצנרת של הליבה:

fetch_stage, decode_stage, execute_stage, memory_stage ו **writeback_stage**- אלו חמשת שלבי הצנרת (Pipeline) של הליבה, אשר כל אחד מהם אחראי על שלב מסוים בביצוע ההוראות. שלבים אלו מתקדמים במחזורי שעון עוקבים, ומאפשרים ביצוע מקבילי של מספר הוראות במקביל.

לניהול המטמון, הליבה כוללת את השדות:

cache_now ו **cache_updated**- מבנים אלו מייצגים את מצבי המטמון הנוכחי והמתעדכן בליבה **cache_now**. מחזיק את המצב הנוכחי של הנתונים במטמון, בעוד **cache_updated** שומר את השינויים שייכנסו לתוקף במחזור השעון הבא. תכנון זה מאפשר עבודה סינכרונית עם רכיבי הבאס והזיכרון הראשי.

לניהול התקשורת עם הבאס, הליבה משתמשת במבנים:

- **requestor**: רכיב אחראי על שליחת בקשות גישה לבאס עבור קריאה או כתיבה לנתונים מהזיכרון הראשי או מטמונים אחרים.
- **snooper**: רכיב שמאזין לפעולות המתרחשות בבאס ומעדכן את מצבי המטמון בהתאם לפרוטוקול MESI, כדי לשמור על קוהרנטיות נתונים בין הליבות.

המבנה כולל גם משתנים לניטור סטטיסטיקות ביצועים:

- **num_read_hits** : מונה את מספר הפגיעות (Hits) במטמון במהלך הסימולציה.
- **num_write_miss** : מונה את מספר ההחטאות (Misses) בעת כתיבה, נתון חשוב לניתוח ביצועים ואופטימיזציה של המערכת.

פונקציות עיקריות בקובץ זה כוללות:

- **configure_core()** : פונקציה זו מאתחלת את כל רכיבי הליבה, כולל הצנרת, הרגיסטרים, המטמון, ורכיבי התקשורת עם הבאס. האתחול מבטיח שהליבה מתחילה במצב נקי ומוכנה לביצוע הסימולציה.
- **advance_core()** : פונקציה זו מקדמת את הליבה למחזור השעון הבא. היא דואגת להעברת נתונים בין שלבי הצנרת, עדכון מצב המטמון, וניהול ההתקדמות של ההוראות בהתאם לתלות בנתונים ולביצועים של רכיבי הזיכרון והבאס.

באמצעות מבנה זה, הליבה מסוגלת לבצע הוראות בצורה יעילה ומקבילית, תוך שמירה על תקשורת רציפה עם הליבות האחרות במערכת. שילוב נכון של הצנרת, המטמון ורכיבי הבאס מאפשר למערכת לעבוד בצורה סינכרונית, ולבצע סימולציה מדויקת של מעבד רב-ליבתי.

3.2.7. decode_stage.h, execute_stage.h, fetch_stage.h, memory_stage.h, writeback_stage.h

הקבצים decode_stage.h, execute_stage.h, fetch_stage.h, memory_stage.h ו-writeback_stage.h מגדירים את חמשת שלבי הצנרת (Pipeline) של ליבה במערכת הסימולציה. כל קובץ מתמקד בשלב ייחודי בצנרת, כאשר השלבים פועלים בסנכרון כדי לאפשר ביצוע מקבילי של הוראות, בדומה לאופן שבו פועל מעבד מודרני. לכל שלב יש תפקיד ברור בתהליך עיבוד ההוראות, החל משליפת ההוראה מזיכרון ההוראות ועד לכתיבת התוצאה בחזרה לרגיסטרים.

- **fetch_stage.h**

קובץ fetch_stage.h מגדיר את שלב שליפת ההוראות (Fetch) בצנרת, שהוא השלב הראשון בתהליך עיבוד ההוראות בליבה. בשלב זה, ההוראה הבאה בתור נשלפת מזיכרון ההוראות הפרטי של הליבה באמצעות מונה התוכנית (PC) Program Counter

פונקציה עיקרית היא **do_fetch_operation()** פונקציה זו אחראית על שליפת ההוראה ממיקום הזיכרון אליו מצביע ה PC. לאחר שליפת ההוראה, מונה התוכנית מעודכן בהתאם, כך שהשלב הבא בצנרת יוכל לעבד את ההוראה הנכונה. במקרים של הוראות קפיצה, פונקציה זו גם מזהה את הצורך בשינוי כתובת היעד.

- **decode_stage.h**

קובץ decode_stage.h מגדיר את שלב פענוח ההוראות (Decode) שבו ההוראה שנשלפה מפורקת לרכיביה. בשלב זה, המערכת מזהה את סוג ההוראה, את הרגיסטרים המעורבים, ואת הערכים המיידיים (Immediate Values).

פונקציה עיקרית היא do_decode_operation() פונקציה זו מפענחת את קוד ההוראה (Opcode) ומזהה את הפעולה שיש לבצע, כמו חישוב אריתמטי, קריאה מזיכרון או קפיצה. בנוסף, הפונקציה מזהה תלויות בנתונים (Data Hazards) - מצבים שבהם הוראה תלויה בתוצאה של הוראה קודמת שעדיין לא הסתיימה. במקרה כזה, הפונקציה מפעילה מנגנון השהיה (Stall) כדי למנוע שגיאות בעיבוד הנתונים.

- **execute_stage.h**

קובץ execute_stage.h מגדיר את שלב הביצוע (Execute) שבו מתבצעת הפעולה המרכזית של ההוראה באמצעות יחידת החישוב האריתמטית-לוגית (ALU). זהו השלב שבו מתבצע החישוב בפועל על סמך הנתונים שהוכנו בשלב הפענוח.

פונקציה עיקרית היא do_execute_operation() - פונקציה זו מפעילה את ה ALU לביצוע פעולות חישוביות וכן פעולות לוגיות. בנוסף, עבור הוראות קפיצה מותנות, הפונקציה בודקת את התנאים ומעדכנת את מונה התוכנית במידת הצורך. במקרה של הוראות גישה לזיכרון, הפונקציה מחשבת את כתובת הזיכרון המדויקת שאליה יש לגשת.

- **memory_stage.h**

קובץ memory_stage.h מגדיר את שלב הגישה לזיכרון (Memory) שבו מתבצעת קריאה או כתיבה לנתונים במטמון או בזיכרון הראשי. זהו שלב קריטי במיוחד במעבד מרובה ליבות, שבו יש צורך בניהול נכון של קוהרנטיות הנתונים.

פונקציה עיקרית היא **do_memory_operation()** - פונקציה זו מבצעת את פעולת הקריאה (Load) או הכתיבה (Store) בהתאם לסוג ההוראה. אם מדובר בהוראת קריאה, הפונקציה בודקת האם הנתון קיים במטמון הליבה (Cache Hit). אם לא, נשלחת בקשת קריאה לבאס, והנתון נטען מהזיכרון הראשי או ממטמון של ליבה אחרת (Cache Miss). במקרה של הוראת כתיבה, הפונקציה מעדכנת את המטמון בהתאם למדיניות Write-Back ומבצעת סנכרון עם הזיכרון הראשי במידת הצורך.

- **writeback_stage.h**

קובץ writeback_stage.h מגדיר את שלב הכתיבה חזרה (Write-Back) שהוא השלב האחרון בצנרת. בשלב זה, התוצאה של ההוראה שנעשתה בשלב הביצוע נכתבת חזרה לרגיסטרים של הליבה.

פונקציה עיקרית היא **do_write_back_operation()** - פונקציה זו כותבת את תוצאות החישוב או הנתונים שנשלפו מהזיכרון לרגיסטרים הרלוונטיים בליבה. לדוגמה, תוצאה של הוראת ADD תישמר ברגיסטר היעד (Rd). במקרים מיוחדים, כמו רגיסטרים שלא ניתן לשנות (כגון רגיסטר אפס), הפונקציה מוודאת שהנתונים לא ייכתבו בטעות. לאחר מכן, ההוראה מסומנת כהושלמה, והצנרת מתפנה לעיבוד הוראות נוספות.

באמצעות חלוקה ברורה זו לשלבי צנרת, מערכת הסימולציה מסוגלת לבצע הוראות בצורה מקבילית, דבר המאפשר שיפור משמעותי בביצועים .כל ליבה יכולה לעבד מספר הוראות במקביל בשלבים שונים של הצנרת, תוך ניהול נכון של תלויות בנתונים והבטחת עקביות המידע. מבנה מודולרי זה גם מקל על תחזוקת הקוד ומאפשר איתור תקלות ושיפור ביצועים בצורה ממוקדת בכל אחד משלבי הצנרת.

3.2.8. file_handler.h

קובץ file_handler.h אחראי על ניהול קבצי הקלט והפלט במערכת הסימולציה. תפקידו המרכזי הוא לטעון את נתוני הקלט הדרושים לתחילת הריצה, כמו קבצי ההוראות והזיכרון, ולשמור את תוצאות הסימולציה בקבצי פלט לצורך ניתוח ובדיקת ביצועים. ניהול נכון של קבצים אלו מאפשר מעקב מדויק אחרי תפקוד הסימולטור ומספק תובנות חשובות על אופן ביצוע ההוראות במערכת.

בין הפונקציות העיקריות המוגדרות בקובץ:

- **openFile()** : פונקציה זו פותחת קבצים לקריאה או כתיבה בהתאם לצורך. היא משמשת לטעינת קבצי קלט כמו קבצי ההוראות (imem.txt) וזיכרון ראשי (memin.txt) וכן לכתיבת פלטים עם תוצאות הסימולציה. הפונקציה מטפלת גם בבדיקות שגיאות, למשל אם הקובץ לא קיים או שיש בעיה בגישה אליו.
- **load_instruction_memory()**: פונקציה זו אחראית על טעינת קבצי ההוראות לזיכרון ההוראות של כל ליבה. קבצי ההוראות מכילים את התוכנית שעל הליבה להריץ, והטעינה המדויקת שלהם לזיכרון ההוראות הפרטי מבטיחה שהסימולציה תתבצע בצורה תקינה.
- **writeRegisterFile()** : בסיום הסימולציה, פונקציה זו כותבת את מצב הרגיסטרים של כל ליבה לקובץ הפלט regout.txt. קובץ זה מאפשר למשתמש לראות את ערכים סופיים של הרגיסטרים לאחר ביצוע כל ההוראות, מה שמסייע בבדיקת נכונות הביצוע של הסימולציה.
- **write_bus_trace_line()** : פונקציה זו מתעדת את כל פעולות הבאס שמתבצעות במהלך הסימולציה, וכותבת אותן לקובץ bus_trace.txt. הרישום כולל את כל הטרנזקציות שבוצעו על הבאס, כמו קריאות (BusRd), כתיבות (BusRdX) ופעולות

ניקוי (Flush). המידע בקובץ זה חשוב להבנת אופן ניהול התקשורת בין הליבות והזיכרון הראשי.

- **write_core_trace_line()** : פונקציה זו עוקבת אחרי כל ההוראות שבוצעו בכל ליבה במהלך הסימולציה, וכותבת את המידע לקבצים core_traceX.txt (כאשר X מציין את מספר הליבה). קבצים אלו מספקים תיעוד מפורט של תהליך עיבוד ההוראות, כולל שלבי הצנרת שעברו ההוראות ותזמון הביצוע שלהן.

באמצעות file_handler.h מערכת הסימולציה מנהלת בצורה מסודרת את כל תהליך הקלט והפלט, מה שמבטיח הרצה מדויקת ואמינה של הסימולציה, תוך אפשרות לניתוח מעמיק של ביצועי המערכת ותפקוד הליבות. קובץ זה מהווה מרכיב חשוב לשקיפות הביצועים והבנת תפקוד המערכת, במיוחד בעת ביצוע אופטימיזציות או איתור תקלות.

4. תוכניות הבדיקה באסמבלי

4.1. תוכנית counter

4.1.1. תיאור כללי של התוכנית

תוכנית counter פותחה לצורך בדיקת תפקוד הסימולטור בניהול גישות מתואמות לזיכרון המשותף, תוך התמקדות במבחני קוהרנטיות מטמון (Cache Coherence) וניהול קונפליקטים בגישה לבאס. בתוכנית זו, כל אחת מארבע הליבות מבצעת קידום של מונה המאוחסן בזיכרון הראשי בכתובת 128. הליבות פועלות בסדר קבוע – ליבה 0 מתחילה את הקידום הראשון, אחריה ליבה 1, לאחר מכן ליבה 2 ולבסוף ליבה 3. המטרה המרכזית בתוכנית זו היא לוודא כי המונה מתעדכן בצורה עקבית ונכונה, תוך בדיקה של תפקוד פרוטוקול הקוהרנטיות MESI, מנגנון ניהול הבאס, ויעילות הצנרת של הליבות.

4.1.2. תהליך העבודה והבדיקה

הקוד לכל אחת מהליבות נטען לזיכרון ההוראות (imem) בהתאם לסדר הריצה. למשל, ליבה 0 מבצעת אתחול של המונה וכתיבתו לכתובת 128 ולאחר מכן משאירה את תור ההרצה לליבה הבאה. הקוד מתבצע בתצורה שבה כל ליבה אחראית רק על פעולה אחת בתורו של המונה. בתום ההרצה, הערך בזיכרון הראשי בכתובת 512 אמור לשקף את סכום כל העדכונים שבוצעו על ידי הליבות.

הבדיקות בוצעו על ידי ניתוח הקבצים שנוצרו במהלך הריצה:

- **regout.txt**- קבצי פלט הרגיסטרים שהראו את הערכים המעודכנים בכל ליבה.
- **memout.txt**- תוצאות סופיות בזיכרון הראשי, שם נבדק שהערך הסופי בכתובת 512 הוא אכן 512.

- **bustrace.txt**- עקב אחרי טרנזקציות הבאס ובחן את השפעתן על קוהרנטיות המטמון.
- **tsram.txt** ו **dsram.txt**- קבצי זיכרון המטמון שתיעדו את המידע השמור במטמונים של כל ליבה.

4.1.3. תוצאות וניתוח

- **נכונות המונה**

ניתוח קבצי הרגיסטרים (regoutX.txt) הצביע על כך שכל ליבה עדכנה את המונה כמצופה. ליבה 0 התחילה עם ערך 509 במונה, ליבה 1 עדכנה ל-510, ליבה 2 ל-511 ולבסוף ליבה 3 סיימה את העדכון בערך 512. הערך הסופי אושר גם בבדיקת memout.txt, שם נמצא הערך 512 בכתובת 512 בזיכרון הראשי, מה שמעיד על הצלחה בביצוע התוכנית ובשמירה על עקביות הנתונים.

- **בדיקת קוהרנטיות המטמון (Cache Coherence)**

תהליך עדכון המונה כלל בדיקה מעמיקה של פרוטוקול MESI. כל כתיבה למונה גרמה לשינוי במצבי הקוהרנטיות של בלוקי המטמון בליבות האחרות. לדוגמה, כתיבה על ידי ליבה אחת גרמה לשינוי מצבי המטמון בליבות האחרות ל-Invalid, מה שמנע גישה לנתונים שאינם מעודכנים. קבצי tsramX.txt הצביעו על כך שמצבי המטמון התעדכנו בהתאם לטרנזקציות בבאס – ליבה 3 שמרה את הערך האחרון במצב Modified, בעוד שבלוקים בליבות אחרות עברו למצב Invalid.

- **תפקוד הבאס וניהול טרנזקציות**

קובץ bustrace.txt תיעד את הטרנזקציות שהתבצעו על הבאס, כולל קריאות (BusRd), קריאות בלעדיות לכתיבה (BusRdX) וכתיבות חזרה לזיכרון (Flush). ניתוח הבאס הצביע על כך שמנגנון ניהול הבאס עובד בצורה תקינה, תוך חלוקה הוגנת של הגישות בין הליבות באמצעות מנגנון Round-Robin. הבאס אפשר גישה רציפה לכל ליבה לפי תורה, ללא קונפליקטים משמעותיים או השהיות מיותרות. גם כאשר התרחשו Cache Misses, הסימולטור טיפל בהן בצורה יעילה.

- **ביצועים וסטטיסטיקות**

קבצי הסטטיסטיקות (statsX.txt) הראו שהליבות חוו השהיות מינימליות בצנרת, בעיקר בשלב הפענוח (Decode Stall) ובשלב הגישה לזיכרון (Mem Stall) ליבה 3, שביצעה את העדכון האחרון, חוותה את מספר מחזורי השהיה הגדול ביותר, אך הפערים בין הליבות היו קטנים, מה שמצביע על יעילות הסימולטור במניעת צווארי בקבוק.

4.1.4. מסקנות

תוכנית counter סיפקה בדיקה אפקטיבית ליכולת הסימולטור לנהל גישות מתואמות לזיכרון המשותף במערכת מרובת ליבות. תוצאות הבדיקה מצביעות על כך שמנגנון הקוהרנטיות MESI מיושם בצורה תקינה, ניהול הבאס פועל ביעילות, ותזמון ההוראות בצנרת הליבות מתבצע בצורה מסודרת ומדויקת. לא זוהו קונפליקטים חמורים בגישה לבאס, והסימולטור הצליח לשמור על עקביות הנתונים בזיכרון

הראשי ובמטמונים של הליבות. הערך הסופי במונה תואם את הציפיות, והמערכת הציגה ביצועים טובים גם במבחני עומס זיכרון.

באמצעות תוכנית זו, הצלחנו לוודא את תקינות המערכת ולבחון את יכולותיה בניהול קוהרנטיות וביצועים במערכת מרובת ליבות.

4.2. תוכנית addserial

4.2.1. תיאור כללי של התוכנית

תוכנית addserial נועדה לבדוק את תפקוד הסימולטור במצבי חישוב סדרתיים (Serial Computation) ולבחון את הביצועים של כל ליבה בעבודה עצמאית על כמויות גדולות של נתונים. כל אחת מארבע הליבות מבצעת סכום של שני וקטורים בגודל 4096 אלמנטים, ללא תלות בליבות האחרות. הווקטור הראשון מאוחסן בכתובות 0 עד 4095 בזיכרון הראשי, והווקטור השני מאוחסן בכתובות 4096 עד 8191. התוצאה של חיבור הווקטורים נשמרת בכתובות 8192 עד 12287. המטרה בתוכנית זו היא לבדוק את יעילות הצנרת של הליבות, גישה לזיכרון, ואת ביצועי המטמון במצבים של עומס גבוה על ליבה יחידה.

4.2.2. תהליך העבודה והבדיקה

הקוד של כל ליבה נטען לזיכרון ההוראות (imem) באופן עצמאי, כך שכל ליבה מריצה את החישוב שלה ללא תלות בליבות האחרות. כל ליבה מעבדת את כל 4096 האלמנטים ומבצעת את פעולת הסכום עבור כל זוג ערכים. לאחר סיום החישוב, הליבות כותבות את התוצאה לאזורים המוגדרים בזיכרון הראשי.

הבדיקות התבצעו באמצעות ניתוח הקבצים שנוצרו במהלך הריצה:

- **regoutX.txt** – מציג את מצב הרגיסטרים לאחר סיום ההרצה עבור כל ליבה.
- **memout.txt** – מציג את המצב הסופי של הזיכרון הראשי ומאפשר לבדוק אם הסכומים חושבו נכון.
- **bustrace.txt** – עוקב אחרי הטרנזקציות שבוצעו בבאס ובוחן את השפעתן על הביצועים.
- **tsramX.txt** ו- **dsramX.txt** – מציגים את מצב המטמון של כל ליבה לאחר ההרצה.
- **statsX.txt** – מספק נתונים סטטיסטיים על השהיות, פגיעות והחטאות במטמון.

4.2.3. תוצאות וניתוח

- **נכונות חישובי הסכום**

בדיקת קובצי הזיכרון memout.txt הצביעה על כך שכל חישובי הסכום בוצעו בצורה נכונה. כל כתובת בזיכרון שבה נשמרה תוצאת הסכום הכילה את הערך הצפוי, דבר המעיד על כך שהחישוב הסדרתי של כל ליבה עבד כמצופה.

- **בדיקת יעילות המטמון**

בדיקה של קובצי המטמון dsramX.txt ו- tsramX.txt הראתה כי במהלך ריצת התוכנית, מרבית הנתונים נשלפו בצורה יעילה מהמטמון, במיוחד לאחר שהנתונים נטענו בפעם הראשונה. עם זאת, בתחילת הריצה חוו הליבות מספר החטאות במטמון (Cache Miss), במיוחד בעת שליפת הנתונים מהזיכרון הראשי.

- **תפקוד הבאס וניהול טרנזקציות**

קובץ bustrace.txt הצביע על כך שמרבית הטרנזקציות בבאס היו קריאות לנתונים מהזיכרון הראשי. מכיוון שכל ליבה עבדה על בלוקים נפרדים בזיכרון, כמעט ולא נרשמו קונפליקטים בגישות לבאס. הניהול העצמאי של כל ליבה הפחית את עומס הבאס והשפיע לטובה על הביצועים.

- **ביצועים וסטטיסטיקות**

קובצי הסטטיסטיקות statsX.txt הראו הבדלים קלים בזמני הריצה בין הליבות, הנובעים מתזמון פנימי בצנרת והשהיות בגישה לזיכרון. ליבות מסוימות חוו השהיות בשלב הקריאה מהזיכרון הראשי, אך באופן כללי, כל הליבות סיימו את החישובים בזמנים דומים, מה שמצביע על איזון טוב בביצועי הסימולטור.

4.2.4. מסקנות

תוכנית addserial הוכיחה כי הסימולטור מתפקד בצורה תקינה גם במצבים שבהם כל ליבה פועלת עצמאית על משימות חישוביות מורכבות. ניהול הזיכרון והמטמון התבצע ביעילות, והמערכת הצליחה לשמור על ביצועים יציבים לאורך כל הריצה. לא זוהו קונפליקטים משמעותיים בגישה לזיכרון המשותף, והביצועים היו עקביים עם ציפיות ממערכת מרובת ליבות.

באמצעות תוכנית זו, ניתן היה לבחון את ביצועי הסימולטור במצבים של חישובים סדרתיים עצמאיים ולהעריך את יעילות ניהול הזיכרון והמטמון במערכת.

9.3. תוכנית addparallel

9.3.1. תיאור כללי של התוכנית

תוכנית addparallel נועדה לבדוק את ביצועי הסימולטור במצבי ריבוי ליבות עם גישה מקבילית לזיכרון. המטרה המרכזית של התוכנית היא לבצע חישוב סכום של שני וקטורים גדולים, בגודל של 4096 אלמנטים, בצורה מקבילית. כל אחת מארבע הליבות אחראית על חישוב חלק מהווקטור, ובכך מנוצלת היכולת של מערכת מרובת ליבות לבצע משימות במקביל. התוכנית מתמקדת במבחן הביצועים (Performance Benchmark), שבו זמן הריצה הכולל של התוכנית הוא המדד העיקרי להצלחה. ככל שהזמן קצר יותר, כך הביצועים טובים יותר.

9.3.2. תהליך העבודה והבדיקה

הקוד נטען לזיכרון ההוראות של כל אחת מהליבות (imemX.txt), כאשר כל ליבה מבצעת חישוב עצמאי של חלק מהווקטור.

- ערכי הווקטור הראשון נטענים מכתובות 0 עד 4095.

- ערכי הווקטור השני נטענים מכתובות 4096 עד 8191.

- תוצאת הסכום נכתבת לכתובות 8192 עד 12287.

כל ליבה מקבלת טווח כתובות שונה, על מנת למנוע התנגשויות בגישה לזיכרון. עם סיום החישובים, נבדקו קבצי הפלט כדי לוודא את נכונות הנתונים ולבחון את ביצועי המערכת:

- **regoutX.txt** – הציג את הערכים הסופיים של הרגיסטרים לכל ליבה.
- **memout.txt** – הציג את התוצאה הסופית בזיכרון הראשי.
- **bustrace.txt** – עקב אחרי טרנזקציות הבאס ובדק את השפעתן על יעילות הגישה לזיכרון.
- **tsramX.txt** ו- **dsramX.txt** – הציגו את מצבי המטמון והנתונים שנשמרו בו.
- **coreXtrace.txt** – עקב אחרי הוראות שבוצעו בכל ליבה.
- **statsX.txt** – נתוני סטטיסטיקה, כולל זמני השהיה, פגיעות במטמון והחטאות.

9.3.3. תוצאות וניתוח

- **נכונות התוצאה**

בדיקה של קובץ memout.txt אישרה שהתוצאה של סכום הווקטורים נכונה, וכל הנתונים נכתבו לכתובות המתאימות בזיכרון הראשי. קבצי regoutX.txt הציגו ערכים סופיים שתואמים את הציפיות עבור כל ליבה.

- **ביצועים ויעילות**

קבצי statsX.txt הראו שיפור ניכר בביצועים בהשוואה להרצה הסדרתית (addserial). זמני ההשהיה ירדו באופן משמעותי, בעיקר בשל חלוקת העבודה בין הליבות. ליבות עבדו במקביל עם מינימום התנגשויות בגישה לזיכרון.

- **קוהרנטיות מטמון**

ניתוח קבצי tsramX.txt ו-dsramX.txt הצביע על כך שמנגנון הקוהרנטיות (פרוטוקול MESI) פעל בצורה תקינה. כיוון שכל ליבה עבדה על טווח זיכרון ייחודי, היו מעט מאוד קונפליקטים במטמון.

- **ניהול הבאס**

קובץ bustrace.txt הצביע על חלוקת גישות לבאס בצורה הוגנת, עם מעט השהיות. מנגנון ה-Round-Robin לניהול גישות לבאס עבד בצורה אפקטיבית ומנע צווארי בקבוק.

9.3.4. מסקנות

תוכנית addparallel הדגימה את היכולת של הסימולטור להתמודד עם חישובים במקביל במערכת מרובת ליבות. התוצאות מצביעות על שיפור ביצועים משמעותי בזכות חלוקת העבודה והפחתת העומס על הבאס. מנגנוני הקוהרנטיות והניהול של המטמון פעלו בצורה תקינה, מה שמאפשר עבודה במקביל ללא טעויות נתונים.

באמצעות תוכנית זו, הצלחנו לוודא את יכולות הסימולטור בביצוע חישובים מקבילים, ולבחון את הביצועים במערכת מרובת ליבות בתנאי עומס שונים.