מבנה המחשב: פרוייקט סוף

05/01/2022

תוכן העניינים

1 פירוט כללי ועקרונות תכנון 2

2 אסמבלר 2
2.1 סד"פ במבט כללי 2
2.2 המרת שורת אסמבלי לשפת מעבד 2
2.3 אתגרים 2
2.3.1 תוויות 2
2.3.2 פקודות word. 2
2.3.3 פרמוט ההוראות 3

3 סימולטור למעבד MIPS 3
3.1 סד"פ במבט כללי 3
3.2 אתגרים 3
3.2.1 פירוש שורות ההוראה מקובץ ההוראות 3
3.2.2 התמודדות עם ערכי immediate בפרסור שורת ההוראה מקובץ ההוראות 3
3.2.3 טיפול בפסיקות 4
3.2.4 טיפול בהתקני קלט-פלט 4

4 תוכניות בדיקה 4
4.1 ציור עיגול על המסך 4
4.2 מציאת המקדם הבינומי של ניוטון $\binom{n}{k}$ 5
4.3 הכפלת מטריצות 4×4 5
4.4 הזזה של המידע בדיסק הקשיח מסקטור לסקטור 5

1 פירוט כללי ועקרונות תכנון

בפרוייקט נתבקשנו לממש מדמה-מעבד MIPS בשפת C, יחד עם אסמבלר תואם אשר מקבל תוכניות אסמבלי ומדמה את פעולות המעבד עד השלמת התוכנית.
העבודה התבצעה בצוות של שלושה סטודנטים, בעבודה קבוצתית ועצמאית לפי מורכבות המשימות.

במהלך העבודה על הפרוייקט, נדרשנו לשמר עשרות קבועים לחלקים שונים בפרוייקט. בכדי להימנע משימוש ב-magic numbers ועל מנת ליצור קוד קריא, החלטנו לתחזק קבצים יעודיים (Constants.h), שהכילו, לדוגמא, את ההמרה בין שם הרגיסטר לבין האינדקס שלו במעבד, או את ההמרה בין ה-opcode של הוראה כלשהי לבין הערך המספרי שלה אותו האסמבלר צריך לכתוב.

---

2 אסמבלר

האסמבלר נדרש להמיר תוכנית אסמבלי לקוד מכונה בפורמט שתואם למעבד שלנו.

2.1 סד"פ במבט כללי

1. האסמבלר מקבל קובץ אסמבלי ופותח אותו לקריאה.
2. כל שורה בקובץ נכתבת למערך שורות, עם התייחסות נפרדת לשורות שמכילות תווית, ולשורות שהן תווית בלבד.
3. במעבר נוסף על מערך השורות, בכל פעם שמופיעה תווית, היא מתורגמת לכתובת ששיכת לה.
4. כל שורה במערך השורות מתורגמת לייצוג ההקסאדצימלי שלה לפי טבלת ה-opcodes ולפי התבנית של הוראת SIMP כפי שניתנה בקובץ ההוראות.
5. כל שורה במערך נכתבת לקובץ הפלט imemin.txt, וכל הוראות ה-word. שנשמרו במערך גם הן נכתבות לקובץ הפלט dmemin.txt.

2.2 המרת שורת אסמבלי לשפת מעבד

עיקר עבודת האסמבלר הוא להמיר את הקוד שכותב המשתמש לשפת המעבד.
תהליך ההמרה בוצע בשני שלבים:

1. פירוק השורה למרכיביה (opcode, רגיסטר rd, ... רגיסטר imm2)
2. סידור כל הערכים לפי פורמט הוראת SIMP

את פירוק השורה ביצענו בעת קריאה השורה מקובץ התוכנית.
את סידור הערכים לפי הפורמט ביצענו באמצעות פונקציית _FormatAsHex שכתבנו, שמקבלת כקלט את פרמטרי השורה המפורקת, ומבצעת את הפעולות הנדרשות בכדי להתאים את המידע שנכתב בתוכנית לפורמט ההוראה.

2.3 אתגרים

2.3.1 תוויות

התמודדות עם labels היה האתגר המשמעותי ביותר בחלק זה של הפרויקט. האסמבלר צריך לדעת להפנות את הוראות הקפיצה למקום המיועד אליהן, בלי לדעת מראש כמה תוויות תוגדרנה ע"י כותב התוכנית.

לצורך התמודדות עם אתגר זה, ביצענו את תהליך התרגום לשפת מכונה בשני שלבים.
בשלב הראשון, עברנו על כל שורה בקובץ האסמבלי ושמרנו אותה במערך שורות. תוך כדי מעבר זה, כשנקלנו בתווית, שמרנו את שמה ואת השורה אליה היא מצביעה בטבלה (לימים, שורה זו תהפוך להיות ה-PC שאליו נפנה את המעבד).
בשלב השני, עברנו על מערך השורות בזיכרון והחלפנו כל הפניה לתווית במספר השורה הרלוונטית מתוך הטבלה. בשיטה זו אנחנו יכולים להתמודד עם כל כמות של תוויות בכל תוכנית אסמבלי שנרצה להעביר באסמבלר.

2.3.2 פקודות word.

פקודת word. דרשה טיפול מיוחד, כיוון שהיא עצמה אינה הוראה עבור המעבד עצמו, אלא הוראה עבור האסמבלר. הוראות אלו נאספו לתוך טבלה שמכילה את ה-offset בזיכרון אליו כותב תוכנית האסמבלי רצה לכתוב, יחד עם שאותו יש לכתוב בזיכרון.
בסוף פעולת האסמבלר שבונה את קובץ הפלט לפקודות, האסמבלר עובר על טבלת ה-word-ים וכותב כל אחד לקובץ הפלט dmemin.txt כנדרש.

2.3.3 פרמוט ההוראות

התאמת הערכים שמופיעים בתוכנית של המשתמש לפורמט ההוראה של מעבד SIMP לא הייתה משימה טריוויאלית כפי שציפינו. התכנון המקורי שלנו היה לשמור את כל הערכים המפורסרים כמספרים שלמים, ולהשתמש בפונקציות לכתיבה מפורמטת של שפת c, פונקציות כמו sprintf.
בסופו של דבר, נעזרנו ב-sprintf עם הפורמט **%02X%02X%02X%02X%02X%02X** שכותב את הערכים שלנו לתוך פורמט של ספרות הקסאדצימליות ברוחב 2 ספרות.

עיקר הסיבוך היה בהתמודדות עם הערכים שהמשתמש מכניס בתור ערכי immediate, כיוון שכמות הביטים שמוקצה לכל אחד מערכי ה-immediate היא 12, שהוא גודל שלא תואם לגודל של טיפוס נתונים סטנדרטי. במהלך הפרסור של קובץ התוכנית שמרנו את הערכים הללו כמספרים שלמים (int) אך בשלב ההכנסה של הנתונים לפורמט נאלצנו לעשות אריתמטיקת shifts ו-masks כדי לוודא שהדאטא שאנחנו מכניסים אכן יכנס לגודל הרצוי של 2 ספרות בכל פעם.

החלטנו לחלק את המילה הסופית ל-6 אלומות, שכל אחת תיוצג ב-2 ספרות הקסאדצימליות, כאשר חלוקה זו לאלומות דרשה את המניפולציה הבאה:

```
// Code snippet from FormatAsHex_, taken from Parser.c

bundle1 = (rd & 0xF) >> 4 | (rs & 0xF);
bundle2 = (rt & 0xF) >> 4 | (rm & 0xF);

// For bundle 3, do this in a few steps:
bundle3 = (imm1_in & 0xFFF) >> 12 | (imm2_in & 0xFFF);

// Multiply by 0xFF here to (only) keep enough information for 2 hex characters.
c1 = (bundle3 << 16) & 0xFF;
c2 = (bundle3 << 8) & 0xFF;
c3 = bundle3 & 0xFF;
sprintf(output, "%02X%02X%02X%02X%02X%02X", opcode, bundle1, bundle2, c1, c2, c3);
```

לאחר מניפולציה זו, יכולנו לשמור את האלומות למחרוזת במערך הפקודות שלנו, ובסופו של דבר לפלוט את תוכן המערך לקובץ הפקודות imemin.

---

3 סימולטור למעבד MIPS

הסימולטור מהווה מודל של מעבד MIPS עם סט הוראות מיוחד, יחד עם התקני קלט-פלט.

3.1 סד"פ במבט כללי

1. טעינת כל קבצי הקלט למערכים, אתחול ה-PC וה-Cycle.
2. קריאת השורה ה-PC ממערך השורות לביצוע ופרסור שלה לאופקוד ולערכי הרגיסטרים (התמודדות מיוחדת עם ערכי immediate).
3. ביצוע ההוראה שמגדיר ה-opcode הרלוונטי.
4. עבור opcode סיום התוכנית, halt, כתיבת כל קבצי הפלט לפני סיום התוכנית.

3.2 אתגרים

3.2.1 פירוש שורות ההוראה מקובץ ההוראות

כיוון שידענו מראש את הגודל של כל מרכיב בשורת ההוראה, וכיוון שעבדנו באופן נחרץ עם מחרוזות, כתבנו מתודה לפיצול מחרוזת לפי אינדקס התחלה ואינדקס סיום.
כך, יכולנו לטעון את ההוראה הנוכחית מקובץ ההוראות ולפרש אותה למרכיביה ע"י חיתוך המחרוזת באינדקסים הנכונים, והמרה מכתיב הקסא-דצימלי למספרים.

3.2.2 התמודדות עם ערכי immediate בפרסור שורת ההוראה מקובץ ההוראות

בעת פירוש ההוראות מקובץ ההוראותת היה עלינו להבין אם המשתמש רצה להשתמש בערך שמתאים ל-immediate ע"י התבוננות ברגיסטרים rd, rs, ו-rt ובדיקה האם אחד (או יותר) מהם מצביע ל-imm1 או ל-imm2. במידה ואכן נמצא שאחד מהם מצביע לערך מיידי, ביצענו השמה של הערך שמופיע **בחלק** של ה-imm בהוראה לתוך **החלק** המתאים לרגיסטר המטרה במחרוזת עצמה.

3.2.3 טיפול בפסיקות

המעבד שלנו נדרש להתמודד עם שלושה סוגים של פסיקות.
בכל מחזור שעון, במידה והוא לא מטפל בפסיקה ברגע זה, המעבד בודק האם עליו להפסיק את פעולתו הנוכחית ולקפוץ לטיפול בפסיקה.

1. **פסיקת שעון** (irq0):
פסיקה זו קוטעת את פעולת המעבד לפי זמן (שמגדיר המשתמש).

2. **פסיקת דיסק קשיח (להוראות כתיבה וקריאה)** (irq1):
הכתיבה והקריאה מהדיסק הקשיח הן פעולות בעלות השהייה זמנית אינהרנטית, כתוצאה מהיותו של הדיסק הקשיח רכיב חיצוני למערכת, שמכיל חלקים מכאניים שנדרשים לזוז ולהגיע לפוזיציה מדוייקת על מנת לאפשר למעבד לקרוא את המידע שהוא רוצה (בסימולציה שלנו, הדיסק הקשיח הוא תוכנתי, ולכן אנחנו מדמים את ההשהייה הזו באמצעות 'בזבוז' מחזורי שעון באופן תוכנתי). המשתמש שכותב תוכנית אסמבלי יכול לבקש לכתוב לדיסק / לקרוא מהדיסק, אך בפועל המעבד לא יוכל להחזיר לו את תוצאת הקריאה או הכתיבה באופן מיידי, ולכן הפסיקה irq1 משמשת לטיפול במידע שנקרא / בעובדה שנכתב מידע. פסיקה זו מתרחשת 1024 מחזורי שעון לאחר שנתקבלה הוראת הכתיבה או הקריאה.

3. **פסיקת מחזור מעבד** (irq2):
פסיקה זו היא הייחודית מבין שאר הפסיקות, בכך שהיא מתרחשת לפי 'מחזורי שעון' קבועים מראש, שמגיעים בתור קובץ קלט לסימולטור. בכדי להתמודד עם פסיקה זו, קראנו את הקובץ irq2in.txt לתוך מערך בזיכרון, ובכל מחזור שעון בדקנו האם עלינו להרים את דגל ה-irq2status לפי ההשוואה בין המחזור הנוכחי לבין המחזורים בקובץ.

3.2.4 טיפול בהתקני קלט-פלט

המעבד מאפשר לכותב התוכנית להשפיע על פעולתם של התקני קלט-פלט באמצעות הוראות out ו-in. כיוון שמימושם התוכנתי של התקנים אלו היה כמעט לחלוטין בלתי-תלוי בשאר המימוש (למעט שימוש ב-PC, בזיכרון או ב-Cycle) החלטנו להפריד את המימוש לקובץ C חדש שנקרא IO.c ולחשוף החוצה רק את הפונקציות הנדרשות.
ההתקנים המאתגרים ביותר למימוש היו הדיסק הקשיח והמוניטור.

- **מימוש המוניטור:**
המוניטור הוא מערך פיקסלים שערך הצבע שלהם נע בין $0 \to 255$, ולכן בחרנו לממש אותו בתור מערך chars בגודל 256×256. בתחילת ההרצה אנחנו מאתחלים את המערך להכיל אפסים בכל המקומות, ובכך מתקבל מסך שחור לחלוטין.
כתיבה למסך נעשית באחריות המשתמש שכותב את התוכנית.

- **מימוש הדיסק הקשיח:**
נתון שהדיסק הקשיח מכיל 128 סקטורים, שכל סקטור בנוי מ-512 בתים (כלומר 128 מילים), ולכן הגדרנו אותו להיות מערך מספרים שלמים בגודל 128×128.

---

4 תוכניות בדיקה

נדרשנו לכתוב 4 תוכניות בדיקה ספציפיות, בנוסף לתוכנית אותן כתבנו במהלך המימוש בכדי לבדוק את המימוש שלנו באופן נקודתי.
תוכניות הבדיקה עזרו לנו למצוא (ולתקן) טעויות ואי-דיוקים במימוש.

4.1 ציור עיגול על המסך

נדרשנו לצייר עיגול לבן מלא במרכז המסך, מכאן - צבע הפיקסלים קבוע וערכו 255. רדיוס העיגול נתון בזיכרון.
זמן הריצה של תוכנית זו הוא גבוה מאוד. המימוש שלנו דורש לבצע פעולות רבות על מנת להבין מה מרחק הנקודה בה אנחנו מתעניינים כעת ממרכז העיגול.
באמצעות לולאות מקוננות, אנו עוברים על כל נקודה במסך (עם אופטימיזציה, ראו בפסקה הבאה) ומחשבים את המרחק בינה לבין מרכז המסך. אם המרחק קטן מרדיוס העיגול הנדרש, אנו מבקשים מהמעבד לכתוב את צבע הפיקסל למקום הנכון במסך.

על מנת להקטין את זמן הריצה, מימשנו אופטימיזציה טריוויאלית.
במקום לעבור בפועל על כל הפיקסלים במסך, נעבור רק על הפיקסלים שנמצאים בתוך ריבוע שגודל צלעו היא כגודל **קוטר** המעגל (בעצם, אנו עוברים על כל נקודה **בריבוע החוסם** של המעגל).
עבור מעגלים קטנים מגודל כל המסך, מדובר בשיפור משמעותי בזמן הריצה.
אופטימיזציה זו תיכשל אם המשתמש מבקש לצייר מעגל שקוטרו גדול מקוטר המסך, אך הנחנו שמדובר בקלט לא תקין, ולכן לא טיפלנו במקרה זה.

4.2 מציאת המקדם הבינומי של ניוטון $\binom{n}{k}$

תוכנית זו דרשה להתמודד עם רקורסיה, התוכנית צריכה להקצות לעצמה מקום ב-stack על מנת לשמור ערכי חזרה וכת ובות חזרה.
בכל קריאה רקורסיבית, התוכנית מקצה לעצמה ארבעה מקומות חדשים במחסנית, בהם היא שומרת את הפרמטרים $n,\ k$, את כתובת החזרה וגם את המשתנה $s0$ שבו תכננו במקור לשמור את תוצאה הקריאות הרקורסיביות. בפועל מספיק להקצות שלושה מקומות במחסנית בכל קריאה ולכתוב את הקוד ללא התייחסות למשתנה $s0$.

בכל שלב רקורסיבי, אנו בודקים אם אנחנו עומדים בתנאים של אחד משני מקרי הקצה האפשריים:

$$k = 0 \quad \text{or} \quad k = n$$

במידה וכן, אנו קופצים לטפל במקרה הקצה (מוסיפים $+1$ למשתנה התוצאה שלנו, $v0$) ואז משחררים 4 משתנים מהמחסנית, כדי להפסיק את הפעולה של התוכנית על זוג הפרמטרים $n,\ k$ שעבורם הוספנו $+1$ לתוצאה. הסרה זו של המשתנים מהמחסנית מביאה אותנו בעצם חזרה לקריאה הקודמת של הפונקציה.

במידה איננו עומדים בתנאים עבור מקרי הבסיס, אנחנו מחשבים את $n-1$ וקוראים לפונקציה שוב עם הפרמטרים $\text{binom}\,(n-1,\ k)$, ואז מחשבים את $k-1$ וקוראים לפונקציה שוב עם $\text{binom}\,(n-1,\ k-1)$.
התוכנית מסיימת את פעולתה לאחר שכל המחסנית מתרוקנת פרט לערכים הראשונים שנשמרו בה, שמובילים אותה לקפיצה לסיום התוכנית.

4.3 הכפלת מטריצות 4×4

תוכנית זו דרשה מאיתנו לכפול שתי מטריצות בגודל 4×4 ולשמור את התוצאה בזיכרון.
המימוש שבחרנו הוא מימוש טריוויאלי עם לולאות מקוננות על מטריצת הפלט, כאשר כל איבר במטריצת הפלט מחושב לפי המכפלה הפנימית:

$$c_{ij} = (a_i \cdot b_j) + (a_{i+1} \cdot b_{j+4}) + (a_{i+2} \cdot b_{j+8}) + (a_{i+3} \cdot b_{j+12})$$

עבור $a \in A_{4\times4},\ b \in B_{4\times4},\ c \in C_{4\times4}$ כאשר A ו-B הן מטריצות הקלט ו-C היא מטריצת הפלט.
החישוב לעיל מחושב מתוך העובדה שהמטריצות שלנו שמורות כרצף של 16 מספרים עוקבים בזיכרון, ולכן את השורות (i) אנחנו מקדמים ב-1 בכל פעם, ואילו את העמודות (j) אנחנו מקדמים ב-4 (אורך שורה) בכל פעם.
בסיום חישוב איבר במטריצת הפלט, התוצאה נשמרת בזיכרון עם offset שגם הוא מתקדם ב-1 כיוון שאנחנו מבצעים את המעברים שלנו ביחס למטריצת הפלט.

4.4 הזזה של המידע בדיסק הקשיח מסקטור לסקטור

תוכנית זו דרשה להתממשק מול הדיסק הקשיח, נדרשנו להעביר את המידע בשמונת הסקטורים הראשונים בדיסק הקשיח סקטור אחד קדימה.
בכדי לא לדרוס סקטורים, החלטנו לבצע את פעולת ההזזה מהסוף להתחלה - כלומר, להזיז תחילה את סקטור 7 לתוך סקטור 8, ואז להזיז את סקטור 6 לתוך סקטור 7, וכן הלאה.
בתוכנית האסמבלי היה עלינו לדאוג שאיננו כותבים או קוראים מהדיסק הקשיח במידה ואינו מוכן.