

סיכום - List scheduling

מטרה: רוצים לבנות תזמון יעיל ככל האפשר של הפקודות של התכנית,

היעילות היא במובן של clock cycles.

ישנם 3 שלבים באלגוריתם:

1. בניית גרף תלויות

בונים גרף מכון ממושקל, כאשר כיוון הקשתות נעשה לפי סוגי התלויות הבאים:

- write after read - **Anti (=false)**
- read after write - **Flow (data / true)**
- write after write - **Output dependencies**

משקל: המשקל על כל קשת הוא latency.

2. משקול הקשתות

רעיון: ממשקלים את הקשתות, לפי הארכיטקטורה או תיאור המכונה, לפי latencies. כלומר:

- (1) אם נתון Elcor, אז נשתמש בנוסחאות שיופיעו בדף הבא
- (2) אם נתון Pipeline אז צריך לחשב כמה שלבים יש לעשות "Shift" ימינה לפקודה הנוכחית כדי שלא תהיינה התנגשויות
- (3) אם נתון Reservation table אז צריך "להחליק" את הטבלאות שמתנגשות אחת על השניה עד אשר אין חפיפה ב-X'ים במשאבים (עמודות) משותפים (לפי הסדר של הפקודות), ולספור כמה "החלקות" ביצענו.

הערה חשובה: בשלב התזמון אם נגיד יש פקודות 1,2 שנמצאות ב **READY LIST**,

ולפקודה (1) אין משאבים פנויים, אבל לפקודה (2) יש משאבים פנויים, אז אפילו ש(2) בעדיפות יותר נמוכה מ-(1), נשלח אותה לביצוע...

הערה חשובה: בשלב התזמון אנו מניחים by default שהחומרה עושה stalls, כלומר הקומפיילר שלנו לא מכניס **no-ops** בין הפקודות. כלומר מתזמנים את הפקודות, ובסוף מוסיפים א'ים בטור של ה **SCHED** לפי הכמות שצריך.

נוסחאות לחישוב Elcor latency

- ❖ Register flow dependence, $a \rightarrow b$
 - » $\text{Latest_write}(a) - \text{Earliest_read}(b)$
- ❖ Register anti dependence, $a \rightarrow b$
 - » $\text{Latest_read}(a) - \text{Earliest_write}(b) + 1$
- ❖ Register output dependence, $a \rightarrow b$
 - » $\text{Latest_write}(a) - \text{Earliest_write}(b) + 1$
- ❖ Negative latency
 - » Possible, means successor can start before predecessor
 - » We will only deal with latency ≥ 0 , so MAX any latency with 0

- ❖ Memory dependences, $a \rightarrow b$ (all types, flow, anti, output)
 - » $\text{latency} = \text{latest_serialization_latency}(a) - \text{earliest_serialization_latency}(b) + 1$
 - » Prioritized memory operations
 - Hardware orders memory ops by order in MultiOp
 - Latency can be 0 with this support
- ❖ Control dependences
 - » $\text{branch} \rightarrow b$
 - Op b cannot issue until prior branch completed
 - $\text{latency} = \text{branch_latency}$
 - » $a \rightarrow \text{branch}$
 - Op a must be issued before the branch completes
 - $\text{latency} = 1 - \text{branch_latency}$ (can be negative)
 - conservative, $\text{latency} = \text{MAX}(0, 1 - \text{branch_latency})$

Branch latency

by default: יש **stall אחד** (delay slot אחד), לכן $\text{latency} = 2$ (משקל הקשת) בין פקודת branch לפקודה העוקבת.

Delayed branch – הברירת מחדל של הקורס!

אם נתון שהbranch הוא **delayed branch** (מסומן ב **d-br**), אזי מכניסים 1 cc delay בין פקודת branch לפקודה שאחריה. ואז ניתן לדחוף פקודות שלא תלויות ב-branch ב **delay slot** במקום:
(1) לדחוף no-op'ים או
(2) שהחומרה תיצר stalls ולא תטפל בפקודות חדשות

סימון: d-br

הערה: אם זה **non-delayed branch**, אז אין delay insertion ואז משקל הקשת בין פקודת branch לפקודה שאחריה הוא 1.

דוגמא:

```
addiu $t1 $t1 1  
addiu $t2 $t2 1  
beq $t2 $t3 label  
*nop
```



```
addiu $t2 $t2 1  
beq $t2 $t3 label  
* addiu $t1 $t1 1
```

הקוד הימני צריך 3 cc, בעוד שהשמאלי צריך 4 cc.

Memory serialization latency

בין שתי פקודות של גישות לזיכרון. לדוגמא:

```
R1 ← [R2]  
R3 ← R1 * R2  
R3 ← [R1]
```

הdefault: הוא 1 stall, 1 no-op.

אם יש גם serialization וגם dependency אז לוקחים את המaximum מבין תלות **Memory dependence** | **Register flow dependence**.

Back edges – כשזה נמצא בתוך loop ויש תלות באיטרציות קודמות.

הערה: יכולות להיות כמה טבלאות, כמה reservation tables לאותו סוג פקודה, ואז אפשר לבחור בטבלה השניה במקום בטבלה הראשונה, במקום שנצטרך להחליק.

By default: אין יותר מטבלה אחת לכל סוג פקודה.

3. מבצעים list scheduling על סמך הגרף תלויות שבנינו

לחלק לזה שני תתי-שלבים:

- 1) בניית הטבלאות
- 2) הרצת האלגוריתם

3.1. טבלת עדיפויות

טבלת עדיפויות – טבלה שמסכמת עבור כל פקודה את העדיפויות שלה

טבלת עדיפויות

| Instruction# | Priorities | | | | |
|--------------|------------|-------|--------------|---------|----------------|
| | Height | Slack | Register use | Uncover | Original order |
| 1 | | | | | |
| 2 | | | | | |
| 3 | | | | | |
| 4 | | | | | |

הסבר על העמודות של העדיפויות

הערה: העדיפויות רשומות משמאל לימין לפי סדר החשיבות, כלומר Height לפני Slack וכך הלאה...

1. **Height** – (לפי יוסי) מחושב עבור כל צומת, ומיג את "משקל המסלול המקסימלי מהצומת לעלה כלשהו" + 1. כלומר בוחנים מהצומת הנוכחי את כל המסלולים שממשיכים ממנו ומסתיימים בעלה כלשהו, ובחרים את המסלול ה"כבד" ביותר (מבחינת סכום המשקלים על הקשתות), ומוסיפים 1.

לפי בילאל (וגם לפי השקפים, עמ' 22):

נסמן ב- L_{max} את הצומת עם ה- L_{start} המקסימלי. אזי ה-Height של הצומת יהיה $L_{max} - L_{start} + 1$, כאשר ה- L_{start} זה עבור כל צומת בנפרד, כלומר עבור הצומת שמחשבים לו את הגובה.

סדר התעדוף: צומת עם גובה יותר גדול יתועדף קודם.

2. **Slack** – אמצעי למדידה של דרגת החופש של התזמון עבור כל צומת. נתחיל בהסבר של שני מושגים בסיסיים:
- 2.1. **Estart** – הזמן הכי מוקדם בו נוכל לתזמן פקודה. נחשב אותו עבור כל צומת באופן הבא:
- אם לצומת אין אב (כמו לדוגמא שורש), אז $E_{start}=0$.
 - **אחרת** – נבחר את E_{start} להיות $\text{MAX}(E_{start}(\text{predecessor}) + \text{latency})$, כלומר עוברים על כל האבות של הצומת, מסתכלים על הסכום של $E_{start}(\text{predecessor})$ וה latency (כאשר ה latency זה המשקל על הקשת בין האב לצומת), ובחרים מבין כל הסכומים האלו את המקסימלי.

- 2.2. **Lstart** – הזמן הכי מאוחר בו נוכל לתזמן פקודה. נחשב אותו עבור כל צומת באופן הבא:
- אם הצומת הוא עלה, אז $L_{start} = E_{start}$.
 - **אחרת** – נבחר את L_{start} להיות $\text{MIN}(L_{start}(\text{successor}) - \text{latency})$, כלומר עוברים על כל הילדים של הצומת, מסתכלים על החיסור של $L_{start}(\text{successor})$ ב- latency (כאשר ה latency זה המשקל על הקשת בין הצומת לבן), ובחרים מבין כל הסכומים האלו את המינימלי.

ה **Slack** יחושב עבור צומת ע"י הנוסחה: $Slack = L_{start} - E_{start}$.

סדר התעדוף: צומת בעל slack יותר נמוך, יתועדף קודם.

3. **Register use** – יחושב באופן הבא. נסמן ב- **In-degree** את: "מס' הקשתות הנכנסות אל הצומת" נסמן ב- **Out-degree** את: "מס' הקשתות היוצאות מהצומת"
- $$\frac{\text{In-degree}}{\text{Out-degree}}$$
- אזי ה register use יהיה ה $\frac{\text{In-degree}}{\text{Out-degree}}$
- הערה**: אם הצומת הוא עלה, אז נניח שה **Out-degree** הוא 1 (כדי שלא נחלק באפס).

סדר תעדוף: צומת בעל Register use יותר גבוה, יתועדף קודם.

4. **Uncover** – ניתן עדיפות לצמתים עם יותר ילדים.
5. **Original order** – לפי הסדר המקורי של התכנית. כלומר אם כל ארבעת העדיפויות שוות עבור זוג צמתים מסוים, אז הסדר המקורי הוא הכלל השובר-שוויון, נבחר לפי סדר הביצוע המקורי של התכנית. כלומר אם פקודות (1) ו (2) הן בעלות אותן עדיפויות, 1 תיבחר קודם.

3.2 טבלת תזמון והרצת אלגוריתם

טבלת תזמון - אחת שעושה את ה list scheduling בהתבסס על טבלת העדיפויות (הטבלה שבנינו ב-3.1)

| T | RU_map | | RL (READY LIST) | SCHED |
|---|--------|------|-----------------|-------|
| | Res2 | Res1 | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |

בונים את הטבלה שרשומה לעיל, כאשר:

- **T** - מציין את נק' הזמן שבה אנו נמצאים.
- **RL (READY LIST)** - אלה הן פקודות שטרם תוזמנו.
- **READY LIST מכניסים פקודה רק כאשר כל שאר הפקודות שהיא תלויה בהן תוזמנו כבר.**
- **(הערה: הפקודות שפקודה תלויה בהן מוצגות בגרף ע"י הקשתות המכוונות שנכנסות לפקודה).**
- **SCHED** - הפקודות שאנו מתזמנים במחזור השעון הנוכחי.

העמודות האופציונליות הן:

- **RU map** - קיצור של Resource usage map, זו מפה שמציינת בכל cycle נתון, אילו - מהמשאבים נמצאים בשימוש (ע"י סימון X במקום המתאים).
- **Res1** - משאב כלשהו.
- **Res2** - משאב אחר כלשהו.

אלגוריתם

1. להוסיף את כל הפקודות בתכנית לתור שנקרא "**UNSCHEDULED**".
2. $time = -1$
3. כל עוד "**UNSCHEDULED**" אינו ריק בצע:
 - 3.1. $time++$
 - 3.2. הוסף לתור ה**READY** את הפקודות אשר מוכנות להתבצע (הערה: ל**READY LIST** אנחנו מכניסים רק אחרי שעשינו scheduling לפקודות שאנו תלויים בהם)
 - 3.3. מיין את תור ה**READY** לפי הטבלת עדיפויות (חמשת העדיפויות)
 - 3.4. עבור כל פקודה בתור ה**READY** בצע:
 - 3.4.1. אם המשאבים שהפקודה צריכה פנויים אזי:
 - תזמן את הפקודה
 - סמן ב"X" את המשאבים שהפקודה תופסת בטבלת המשאבים
 - הסר את הפקודה מתור ה"**UNSCHEDULED**" ותור ה"**READY**"
 - 3.4.2. אחרת, המשך

הערות, הנפצות ושאר ירקות

(1) טבלאות משאבים - אם נתונים 2 סוגים של פקודות, A ו B, ופקודה (1) היא מסוג A, ופקודה (2) היא מסוג B, ונתון $latency(1,2)=3$ אז ניתן להניח בבניה של הטבלאות משאבים ש $latency(A,B)=3$ עבור כל פקודה מסוג A וכל פקודה מסוג B.

(2) Pipeline – האורך של ה pipe הוא אחיד, כלומר מתחילת pipe ועד סופו זה אותו מספר שלבים עבור כל סוגי הפקודות. כלומר **IF ID EX MEM WB**, ויתכן שב **EXE** נבלה מס' שונה של מחזורי שעון עבור פקודות שונות. תיתכן יחידה אחת שעובדת ב latencies שונים, כמו **ALU** שעובד עבור **3 שלבים** ועבור חיבור **1 שלבים**. אבל אז אם יש חיבור אחרי כפל אז נעשה ביניה $latency=4$.

כמות יחידות:

- יכולות להיות כמה יחידות של EXE + MEM (אם שתי פקודות ניגשות לאותה כתובת בזיכרון, ניתן להניח שהאחרונה ביותר כתבה).
- לא יכול להיות **2 WB**.
- ב **pipeline** רגיל יש יחידה אחת מכל סוג, **DECODE** ו **FETCH** וכך הלאה...

(3) יש 2 מודלים בקורס של stalls: (כאשר יש התנגשות בין משאבים)

- **NO-OP**, כאשר ה compiler מייצר stalls ע"י פקודות **NO-OP**
- **STALL** ע"י החומרה.

defaultn: זה stall ע"י החומרה.

- יש שתי אפשרויות לקלטים של המכונה והתכנית:
 - דגם 1: יש טבלאות משאבים, ואז מוסיפים עמודה בשם **RU_map**, ואז ממלאים את ה **ready list** ועושים **sched**.
 - דגם 2: אין טבלאות משאבים, ואז ממלאים את ה **ready list** ועושים **sched**.

- אם נתון בשאלה $latency(1,2)=4$ זה אומר שאם רוצים לשבץ פקודה מסוג (2) לאחר פקודה מסוג (1), אז ה $latency$ (המשקל על הקשת) יהיה 4.

- אם נתון בשאלה $latency(2,1)=4$ זה אומר שאם רוצים לשבץ פקודה מסוג (1) לאחר פקודה מסוג (2), אז ה $latency$ (המשקל על הקשת) יהיה 4.

(4) Critical path – יוגדר כך:

$\max(\text{longest path of all nodes without predecessors} + \text{execution time of last instruction})$

כלומר מסתכלים על כל הצמתים שאין להם אבות קדמונים (שורשים), בוחרים עבור כל צומת כזה את המסלול המקסימלי (המסלול הכבד ביותר מצומת לכל אחד מהעלים) ומחברים לו את זמן הביצוע של העלה, ובוחרים את המקסימום מבין כל המסלולים המקסימליים. הערה: critical paths יתכנו כמה

הגדרה אלטרנטיבית:

Critical Path

❖ Critical operations = Operations with slack = 0

- » No mobility, cannot be delayed without extending the schedule length of the block
- » Critical path = sequence of critical operations from node (with no predecessors) to exit node, can be multiple crit paths

(5) כאשר רוצים לחשב חסם תחתון ועליון לזמן ריצה של גוף הלולאה אזי

החסם התחתון: יקבע לפי ה critical path.

חסם עליון: בוחרים schedule סדרתי, וגם כל תזמון אפשרי (חוקי) הוא חסם עליון. כדי להגיע ללו"ז אופטימלי: יש לדחוף כמה שיותר פקודות ב critical path כדי שינצלו את stalls שנגרמים (אם יש כאלה)

(6) ניתן להניח שיש full bypass, כלומר אם ל-pipeline שלנו יש p שלבים, אזי יש לנו מנגנון forwarding שיכול להעביר מידע ל-p-1 שלבים אחורה.

(7) העברת מערכים סטטיים ודינמיים לפרוצדורות

קודם כל, לא יתכן גם מערכים סטטיים וגם דינמיים בתכנית (עפ"י יוסי). אם אנחנו ב mode של מערכים סטטיים אז אין descriptors. אם אנחנו ב mode של מערכים דינמיים אז יש descriptors.

מערך סטטי:

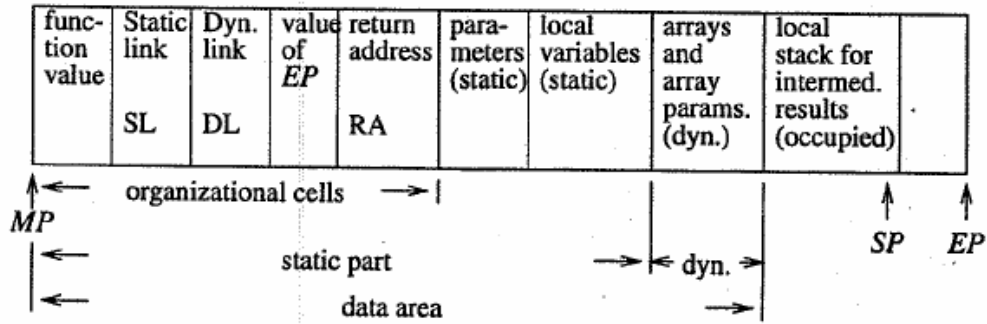
- **אם הוא מועבר by value** - אז הפונקציה הקוראת מעבירה ב `codeA` את תוכן המערך עצמו, כלומר `movs q`, כאשר `q = <size of array>`.
הערה:הsyntax של `movs q` הוא: `movs q`
כאשר `movs` מצפה שבראש המחסנית תופיע כתובת זיכרון, ואז היא מעתיקה `q` מקומות לראש מחסנית החל מהכתובת הנתונה.
- **אם הוא מועבר by reference** - אז הפונקציה הקוראת מעבירה ב `codeA` מצביע לתחילת המערך.
זכרו: במערכים סטטיים אין `descriptor` (אלא אם כן נאמר אחרת).

מערך דינמי:

- **אם הוא מועבר by value** - אז הפונקציה הקוראת ב `codeA`, מעתיקה את `descriptor` של המערך הדינמי אל הפונקציה הנקראת.

חשוב מאוד:

- (1) בכניסה לפונקציה **הנקראת**, היא מבצעת פקודה שנקראת `movd` (move dynamic) שמעתיקה את תוכן המערך לשטח הלוקאלי שלה, ומעדכנת את `descriptor` שהיא קיבלה, כך שה**adjusted array address** שמופיע שם יצביע לגוף המערך שהועתק (כלומר לכתובת תחילת העותק הלוקאלי של גוף המערך).
- (2) **הערה:** הsyntax של `movd` הוא: `movd q` כאשר `q` זה `offset` של `descriptor.array`.
- (3) הפונקציה הקוראת תיקח בחשבון בחישוב ה `sep` את גודל ה- `descriptor`. הפונקציה הנקראת תיקח בחשבון בחישוב ה `ssp` את גודל ה- `descriptor` (לא כולל את גוף המערך שכן הוא דינמי, ורק בכניסה לפונקציה בזמן ריצה ניתן לדעת את גודלו).
הגוף עצמו של המערך לא נכלל בחישוב ה `sep/ssp`, כי לא יודעים מה יהיה הגודל.
3.1 ה `descriptor` נמצא בחלק של ה `parameters(static)`
3.2 הגוף של המערך יושב ב- `arrays and array params. (dyn.)`



- **אם הוא מועבר by reference** - אז הפונקציה הקוראת ב codeA, מעבירה מצביע ל descriptor של המערך הדינמי.
 הערה: כאשר הפונקציה הנקראת תרצה להשתמש במערך (במקרה של by ref), היא תשתמש ב `lod a d <offset>, ind i`, כי כעת מה שיש לנו זה רק מצביע ל descriptor, לכן צריך לעשות dereference כדי לקבל את הכתובת האמיתית שלו. כל זאת נעשה לפני שמיצרים קוד לגישה למערכים דינמיים.

8) מכניזם להעברת פרמטרים לפונקציות

- אם מעבירים ערך של משתנה:
 - **By value** – אז הפונקציה הנקראת יוצרת עותק של הערך שהועבר, ובזמן הביצוע הפונקציה הנקראת וגם לאחר היציאה ממנה, השינוי לא ישתקף כלפי חוץ, כלומר הפונקציה הקוראת (והסביבה שלה) לא יראו את השינויים שבוצעו על המשתנה.
- **By var** – כל שינוי שנעשה לפרמטר בתוך הפונקציה הנקראת, ישתקף מיידית גם לפונקציה הקוראת. לדוגמא:

```
x: FIXED;
Procedure p(i: FIXED by var)
{
    i = 10;
    i = 20;
}
X=5;
P(x);
```

כאשר תבצע השורה:

i = 10

אז הערך של x יהיה 10, כלומר כל עדכון ישתקף כלפי חוץ.

- **Copy-in Copy-out / Copy-by-value-result** – כאשר מעבירים פרמטר בשיטה זו, הפונקציה הנקראת יוצרת עותק של הפרמטר, וביציאה ממנה מעתיקה את הערך של העותק חזרה למשתנה שהועבר.

```
x: FIXED;  
Procedure p(i: FIXED by var)  
{  
    i = 10;  
    i = 20;  
}  
X=5;  
P(x);
```

אזי כאשר $i=10$ מתבצע, X עדיין יהיה שווה ל 5 , אבל ביציאה מהפונקציה p , הערך של x יהיה 20 .

- **Call by name** – קורה בשפות תכנות פונקציונליות, נקרא גם **lazy evaluation**, היא שיטה שבה דוחים את שערור הביטוי שמועבר לפונקציה עד לזמן שנצטרך אותו. לקטע הקוד שיבצע את שערור הביטוי (בתוך הפונקציה נקראת) קוראים **Thunk**. הערה: ה **Thunk** הוא "Pointer to evaluation code and pointer to caller's frame".

9) אופטימיזציות

Volatile – אם שמים את זה לפני משתנה, אז הקומפיילר לא יבצע אופטימיזציות Load-store elimination על המשתנה, כלומר זה משתנה שיכולים לגשת כמה מקורות באופן בלתי צפוי (multi-processing למשל). דוגמא למשתנה כזה: hardware register.

סוגי אופטימיזציות

1) IF PROPAGATION



שיפור: חסכון בגישה לזיכרון, אם לדוגמא $x=2$ אזי בקוד השמאלי ניגש 4 פעמים לזיכרון, ובקוד הימני ניגש 2 פעמים לזיכרון.

הערה: זה גם נקראת אופטימיזציה אריתמטית.

2) אופטימיזציות שקשורות לקוד

2.1 LOAD-STORE-ELIMINATION

הגדרה: השמטת פקודות מיותרות של load/store מהקוד.

דוגמא:

```
STORE 100,R1 // MEM[100] = R1  
LOAD 100,R1 // R1 = MEM[100]
```

אז הקומפיילר יכול להשמיט את הLOAD, כי עשינו כבר STORE מתוך R1 לכתובת הזו, לכן לא צריך לקרוא ממנה שוב (את אותו תוכן).

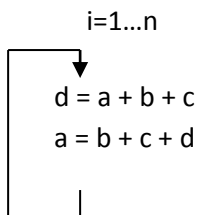
2.2 החלפת LOAD-STORE ע"י שימוש באוגר

הגדרה: במקום להשתמש בכתובות בזיכרון בשביל לאחסן מידע, משתמשים באוגרים.

2.3 הוצאת ביטויים קבועים מלולאות

דוגמא:

```
a = A[1]  
b = A[2]  
c = A[3]
```



שיפור:

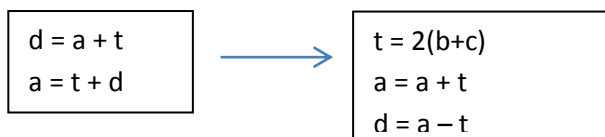
(1) b ו c לא משתנים בתוך הלולאה לכן נחליף אותם בקבועים

נוסיף משתנה $t = b + c$

$d = a + t$

$a = t + d$

(2) הוצאת ביטויים קבועים מלולאות



2.4. הפצת קבועים

להחליף שם משתנה בקבוע, כמו לדוגמא אם היה את הקוד: $y = 0$

ולא כותבים ל y אחרי הפקודה הזו, אלא רק קוראים ממנו, אז אפשר להחליף את y בקבוע 0.



2.5. copy propagation

משמיטים את הפקודה $x=x$



2.6. dead-code elimination

קוד מת, אם אין שימוש ב- x אז אפשר להשמיט אותו מהקוד.



2.7. loop invariant removal

הגדרה: הפעלה של common-subexpression-elimination בתוך לולאה.

2.7. strength reduction

הגדרה: החלפה של פעולה יקרה בפעולה זולה.

דוגמא: כפל בחיבור, או כפל בפעולת ביטים.



Loop unrolling (10)

הגדרה: מעתיקים את "גוף הלולאה" כמה פעמים, פורשים את הלולאה לפי ה unroll factor.

יתרונות:

1. פחות פקודות קפיצה
2. יתכן שנשיג פחות LOAD אם יש לנו תכנית שניגשת הרבה לזיכרון, נוכל להשתמש ברגיסטר שישמור את הערך.

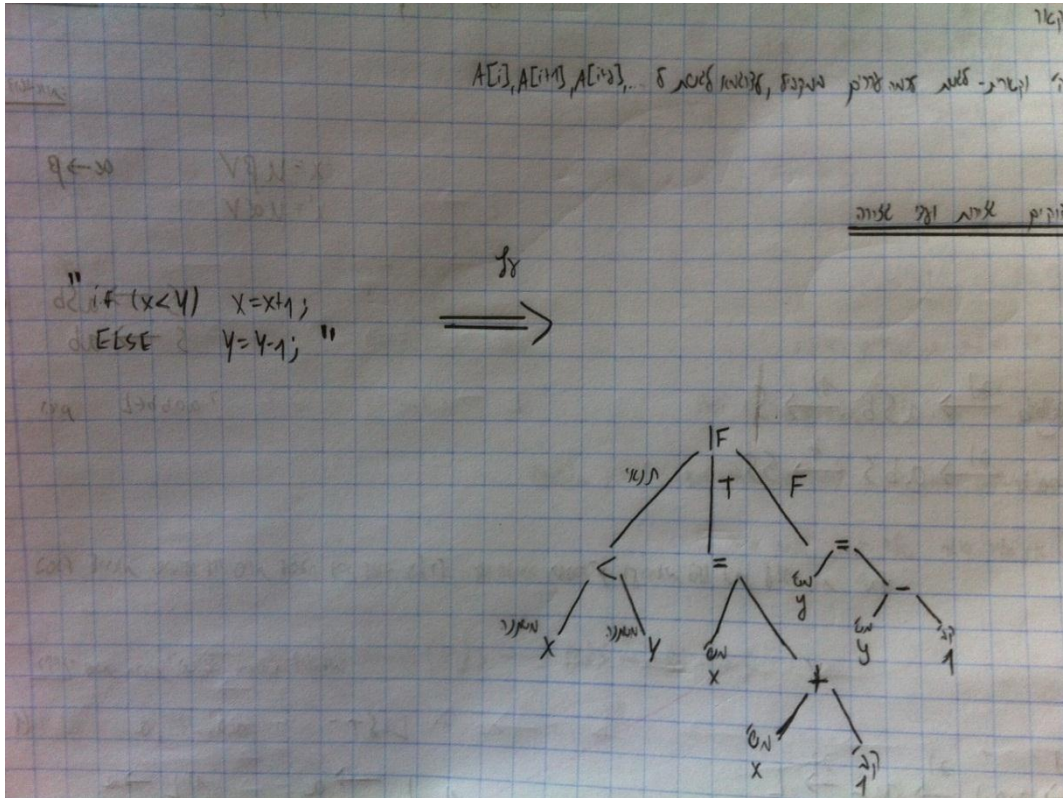
חסרונות:

1. אם עושים unroll factor גדול, אז כמות האוגרים גדלה
2. Instruction cache נפגם, ככל שיש cache קטן יותר יש פחות מקום לפקודות.

(11) **ויקטור** (vectorizing) – פקודה וקטורית שכותבת/קוראת כמה ערכים במקביל, $A[i], A[i+1], A[i+2], \dots$

10) דקדוקים גזרות ועצי גזרה

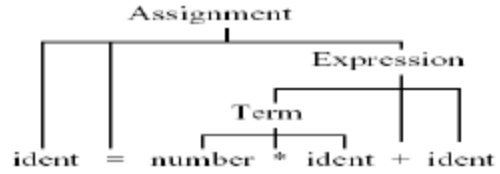
דוגמא:



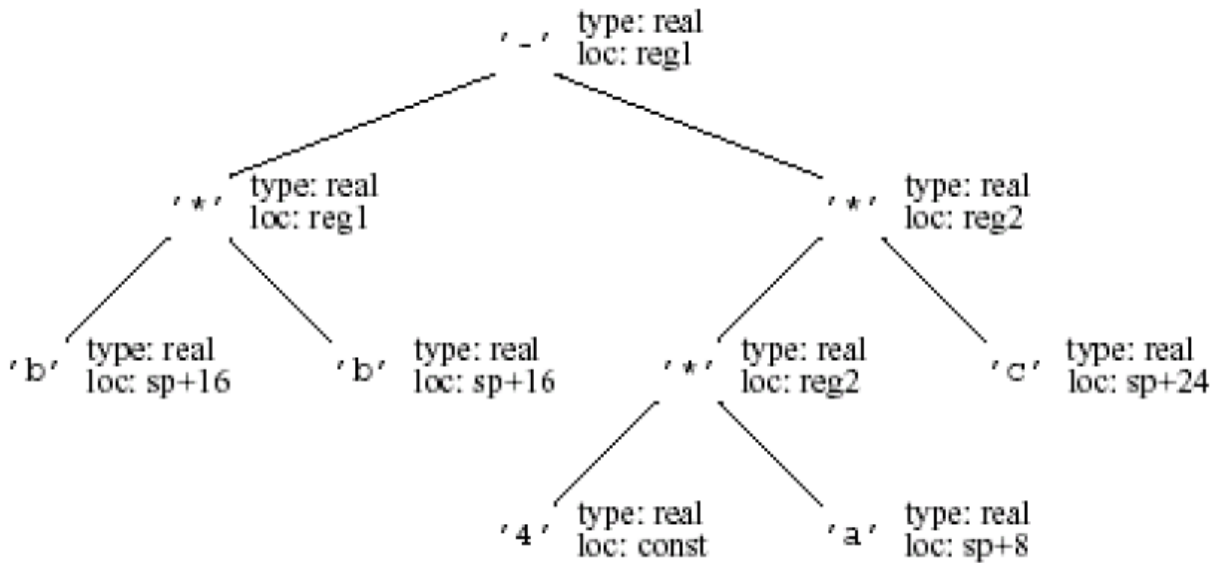
Abstract Syntax Tree (AST)

הגדרה: זה עץ שמתאר את התכנית, דוגמא:





Decorated/Annotated AST – כמו AST, רק שמכיל allocation של אוגרים, וסוגי משתנים.



מיני שפה

בשפה יש 2 סוגי פקודות:

- (1) L – נון-טרמינל התחלתי, התכנית
- (2) S – פקודות (שגוזר A או I)
- (3) A – השמות
- (4) I – תנאים
- (5) E – ביטויים

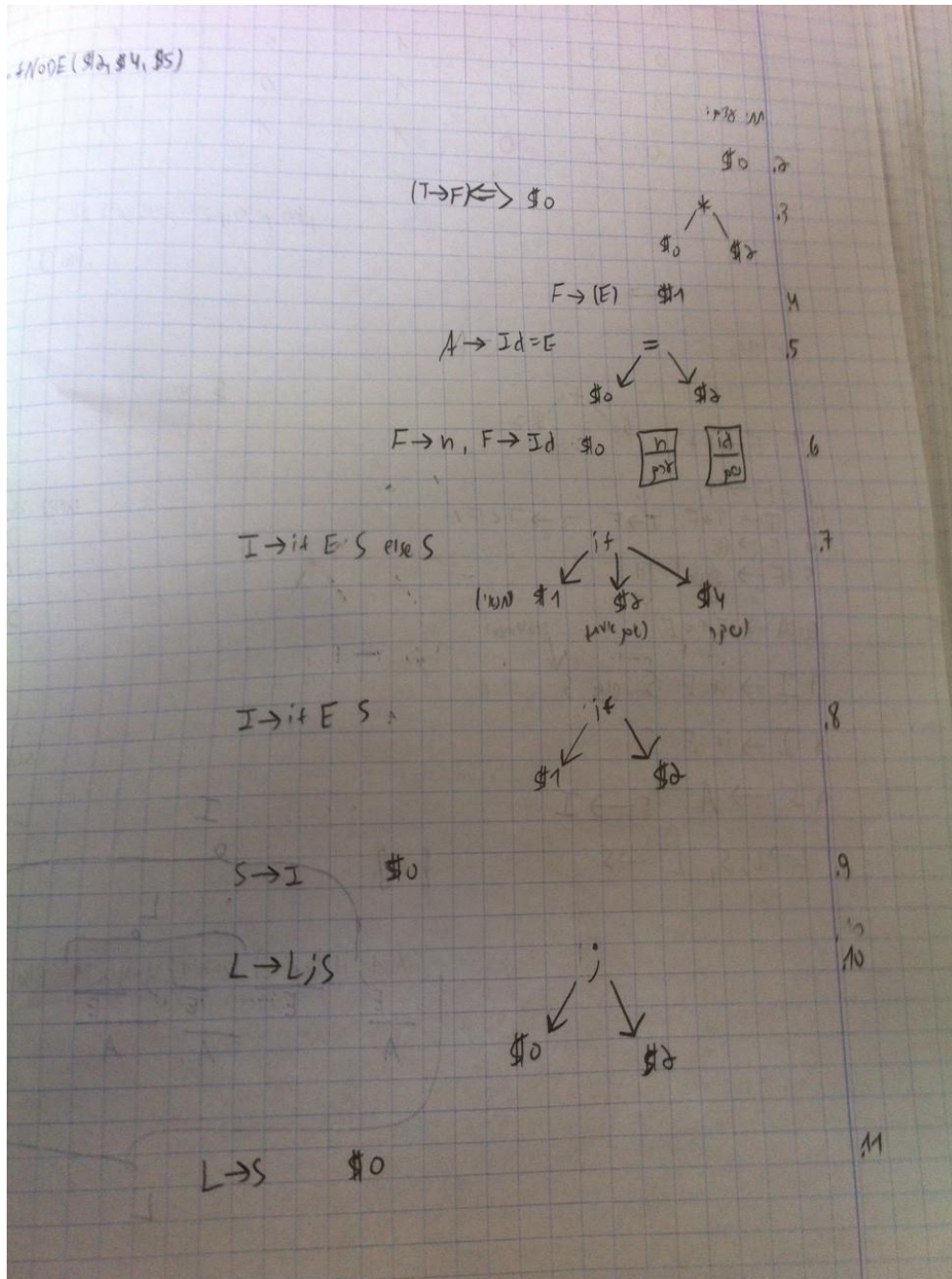
כללי גזירה:

1. $L \rightarrow S \mid L;S$
2. $S \rightarrow A \mid I$
3. $E \rightarrow E+T \mid T$
4. $T \rightarrow T * F \mid T < F \mid F$
5. $F \rightarrow (E) \mid \text{num} \mid \text{ID}$
6. $A \rightarrow \text{Id} = E$
7. $I \rightarrow \text{if } E \text{ S else S}$
 $I \rightarrow \text{if } E \text{ S}$

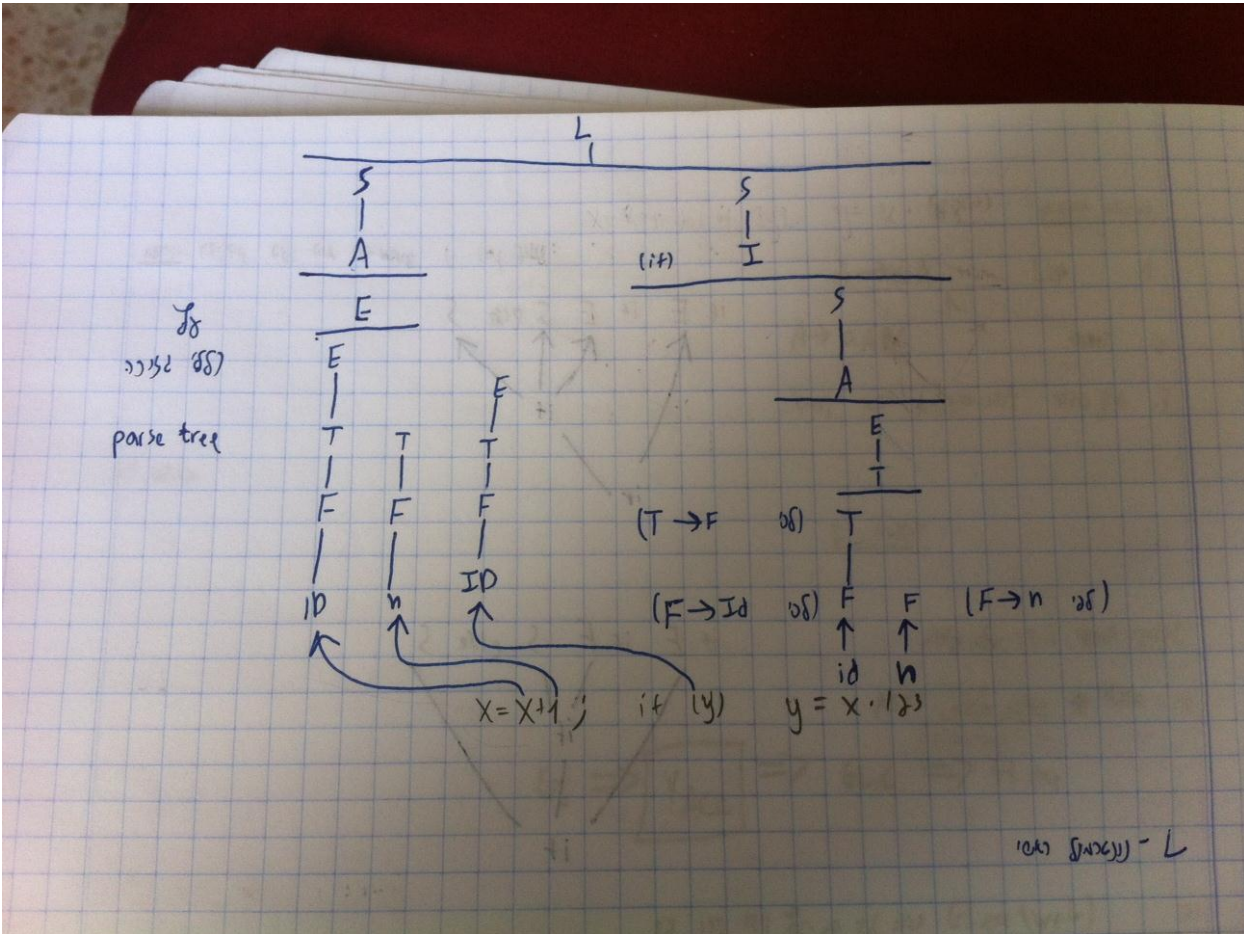
הערה: כנראה שהכוונה ב-כלל (7) זה:

- $$I \rightarrow \text{if } E \text{ L else L}$$
- $$I \rightarrow \text{if } E \text{ L}$$

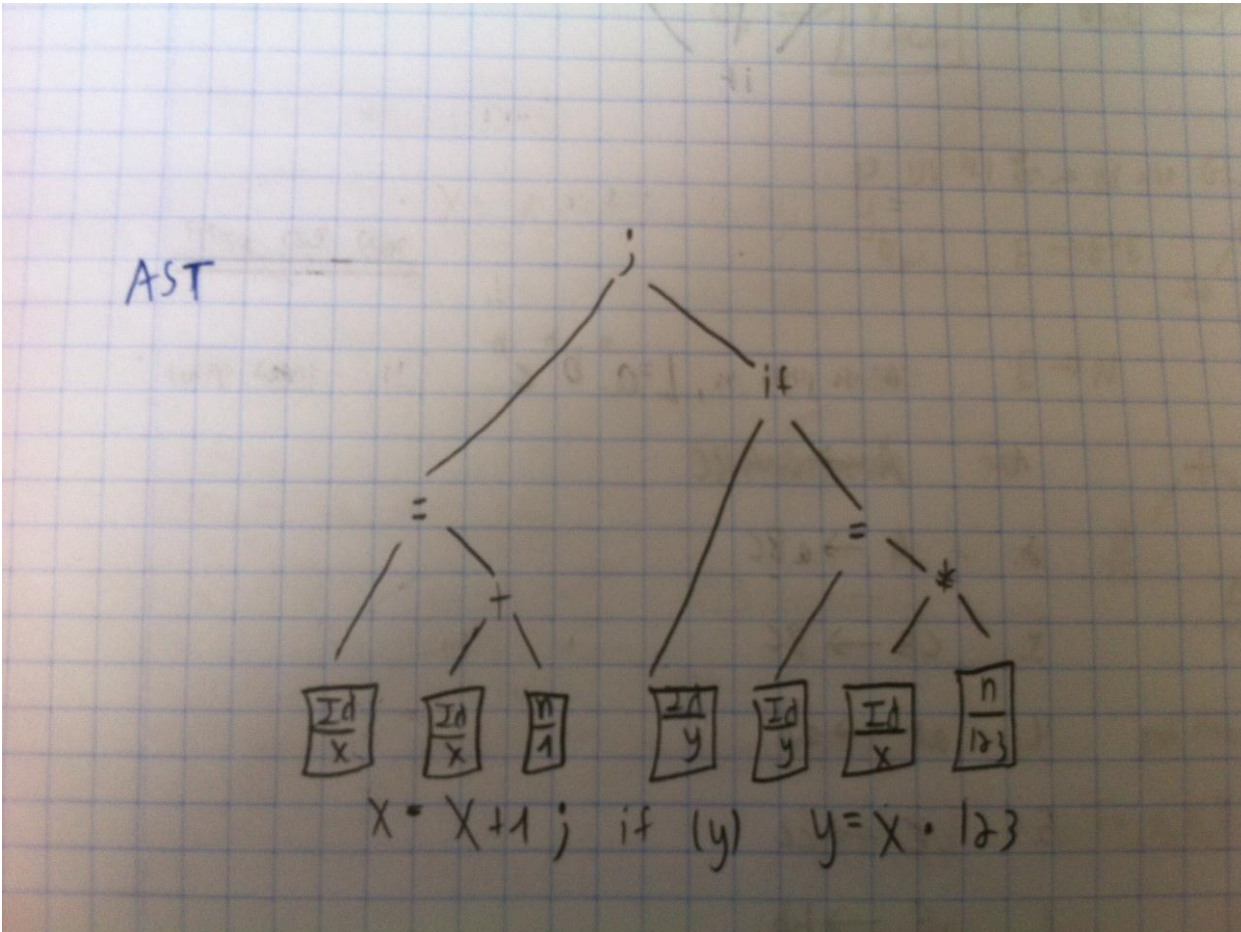
תתי עצים



Parse tree



AST



Register renaming

רעיון: משנים את השם של האוגרים שאליהם כותבים (destination).
מתי אפשר לעשות:

ניתן להפעיל register renaming ברמת ה compiler רק אם נתונה לנו כמות האוגרים במכונה, והתכנית לא משתמשת בכלום, אז ניתן לעשות renaming ל destination registers בהנחה שזה לא משנה את הערך של התכנית.

לרשום למבחן: בהנחה שזו כל התכנית, ולא יהיה שימוש נוסף בערך של האוגר אז אני נפטר מהאוגר כדי להיפטר מהתלות.

מוטיבציה: ישנן 2 סוגים של תלויות:

1) תלויות אמיתיות – Flow (Read After Write), שהן לא ניתנות לפיתרון בלי לשים מרווח מספק בין פקודות תלויות. לדוגמא:

$R1 = \text{MEM}[100]$

$R4 = R1 + R5$

במודל של pipeline יחיד (פשוט) יהיה $latency=2$.

(2) תלויות לא-אמיתיות – תלויות שקורות בגלל ה **out-of-order**, כלומר סדר התזמון

של הפקודות בתכנית הוא לא בהכרח הסדר המקורי של התכנית.

פתרון: **register renaming**, יוסבר עבור כל תלות בנפרד.

ישנם שני סוגים:

(2.1) **Anti** – נקראת גם **Write After Read**, כאשר כתבים לdestination

אחרי שקראנו ממנו.

דוגמא:

$$R1 = R2 + R3$$

$$R2 = 10$$

עבור כל אוגר שאנו רוצים לכתוב אליו, מקצים אוגר פיזי חדש,
לדוגמא עבור **R1** יוקצה האוגר **R14**, ועבור **R2** יוקצה האוגר **R15**,
ונקבל את התכנית הבאה:

$$R14 = R2 + R3$$

$$R15 = 10$$

כעת נפתרה התלות.

(2.2) **Output** – נקראת גם **Write After Write**, כאשר כתבים לdestination

אחרי שקראנו ממנו.

דוגמא:

$$R1 = R2 + R3$$

$$R1 = 10$$

נקבל את התכנית הבאה:

$$R14 = R2 + R3$$

$$R15 = 10$$

כעת נפתרה התלות.