יצוג מספרים שלמים

היצוג של מספרים שלמים במחשב הוא בינארי, כלומר הביטים משמשים כמקדמים של פולינום של חזקות של 2. לדוגמא, מספרים שלמים בני byte אחד (8 ביט) הם

00000000 = 0
00000001 = 1
00000010 = 2
00000011 = 3
00000100 = 4
00000101 = 5
00000110 = 6
00000111 = 7

וכו'. מספר שלם ניתן להתיחס אליו כמספר עם סימן (עשוי להיות שלילי) או חסר סימן (מספר אי שלילי בכל מקרה). במידה והמספר הוא עם סימן ושלילי הוא מיוצג ע"י שיטת המשלים ל-2 (Two's Compliment). בשיטה זו הביט המשמעותי ביותר משמש ביט סימן (0 - חיובי, 1 - שלילי). היצוג השלילי של מספר (חיובי) מסוים מתקבל ע"י היפוך כל הסיביות של המספר החיובי וקידום ב-1. למשל היצוג שי 5- יהיה:

היצוג של +5	=	00000101
אחרי היפוך סיביות	=	11111010
היצוג של -5 = אחרי קידום ב-1	=	11111011

כאשר מתיחסים למספרים בני 8 ביטים כמספרים <u>חסרי</u> סימן, ניתן ליצג את המספרים 255 ... 0. כאשר מתיחסים למספרים בני 8 ביטים כמספרים <u>עם</u> סימן, ניתן ליצג את המספרים 127+ ... 128-.
ב-16 ביט המספרים הטווחים הם 65535 ... 0 ו- 32767 ... 32768- בהתאמה.

מאחר והיצוג של מספרים באוגרי המחשב הם בעלי אורכים מוגדרים (8, 16 ו-32 סיביות) יכולים להווצר בעיות גלישה.

לפעמים סכום שתי מספרים חסרי סימן נותן תוצאה החורגת מהטווח (למשל עבור מספרים 8 ביט, 9 + 250). במקרה הזה התוצאה בגודל ביט אחד יותר מהאופרנדים. למשל בדוגמא:

```
 11111010  =   250
+
 00001001  =     9
 --------
100000011  =   259 (256 + 3)
```

במצב זה נאמר ש-תוצאת הסכום הוא 00000011 וה-carry שווה ל-1.

עבור מספרים בעלי סימן, סכום שני מספרים חיוביים יכול לצאת שלילי, למשל

```
01110000  =  112
+
00101000  =   40
--------
10011000  =  -104
```

מצב כזה נקרא Overflow. מצב דומה, שבו סכום של שני מספרים שליליים הוא חיובי נקרא לפעמים Underflow ב-8086 נחשב לעוד מקרה של Overflow. למשל:

```
10000110 = -122
+
10000011 = -125
--------
00001001 =  +9
```

אמנם יש Carry במצב הזה אבל הוא לא מיוצג בתוצאת הסכום.

בקרה Control

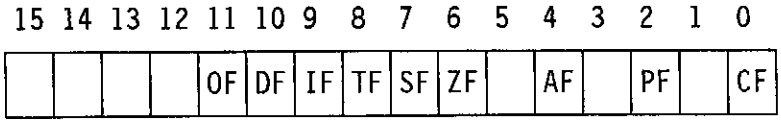
מה שעושה את המחשב או אמצעי הקרוי תכנות לדבר הכללי שהוא הוא היכולת לבצע קוד באופן מותנה ולחזור על ביצוע קוד באופן מותנה. הכוונה לתכנות מבני נוסח if ... else, לולאות נוסך for, while וכדומה. ישנם מודלים מתימטיים לתוכניות שאין את האפשריות הללו (יש למודלים הללו חשיבות בניתוח חישובים אלגבריים נוסך אריתמטיקה של מספרים שלמים, כפל מטריצות...) ומסתבר לתוכניות כאילו הם מוגבלות ביותר מבחינת מה שהן יכולות לעשות.

המימוש של היכולת הזו לבצע קוד באופן מותנה נעשה ע"י 3 גורמים

בחומרה: אוגר הדגלים, פקודות אריתמטיות ופקודות הסתעפות מותנות.

באו נזכר בתאור של אוגר הדגלים:

אוגר הדגלים ב-8086 Flags Register



להלן רשימת הדגלים וקצת תיאור שלהם.

שם הדגל	משמעות לערך 1	סוג הדגל
CF	carry	אריתמטי
PF	זוגיות	אריתמטי
AF	carry עשרוני	אריתמטי
ZF	תוצאה 0	אריתמטי
SF	תוצאה שלילית	אריתמטי
TF	אפשור פסיקה לצורך מימוש Debugger -ים	תפקוד CPU
IF	אפשור פסיקות	תפקוד CPU
DF	כיוון פעולות מחרוזת	תפקוד CPU
OF	גלישה אריתמטית	אריתמטי

מה שרלונטי לדיון שלנו - בקרה או קוד מותנה - הם הדגלים CF, PF, ZF, SF, ו-OF:

שם הדגל	משמעות לערך 1	סוג הדגל
CF	carry	אריתמטי
PF	זוגיות	אריתמטי
ZF	תוצאה 0	אריתמטי
SF	תוצאה שלילית	אריתמטי
OF	גלישה אריתמטית	אריתמטי

הדגלים הללו הם הדגלים האריתמטיים: בתום כל פקודת מכונה אריתמטית נוסח ADD, SUB, MUL, CMP החומרה בודקת את התוצאה של הפעולה, ועל פיה קובעת את

ערכי הדגלים. קביעת ערכי הדגלים היא חלק מתפקידם של הפקודות הללו. למשל:

```
MOV AX,9
MOV BX,9
SUB AX,BX
```

בתום ביצוע פקודת ה-SUB הערך של הדגל ZF יהיה אחד (1) משום שתפקידו של ZF הוא לשקף את התשובה לשאלה "האם התוצאה של הפעולה האריתמטית האחרונה היתה אפס?" והתשובה היא "כן" בדוגמא לעיל. בצורה דומה SF = 0 (אפס) בדוגמא לעיל משום ש-SF צריך לשקף את התשובה לשאלה "האם תוצאת הפעולה האריתמטית האחרונה היתה שלילית?". בדוגמא לעיל התשובה היא לא, כי התוצאה היתה אפס.

בצורה דומה, עבור המקרה

```
MOV AX,9
SUB AX,6
```

הערכים של SF ושל ZF אחרי ביצוע פקודת ה-SUB יהיו SF = 0 ו- ZF = 0.

עבור המקרה

```
MOV AX,9
SUB AX,17
```

הערכים של SF ושל ZF אחרי ביצוע פקודת ה-SUB יהיו SF = 1 ו- ZF = 0.

הדגלים האחרים (CF, OF, PF) מושפעים בצורה דומה. למשל אם נבצע

```
MOV AL,125
ADD AL,12
```

אזי ה-OF יהיה OF = 1.

בכל פקודה אריתמטית כל הדגלים נקבעים.

למשל בפקודה האחרונה, ZF = 0, CF = 0, SF = 1, PF = 0 ו- OF = 1.

עבור המקרה,

```
MOV AL,250
ADD AL,9
```

ZF = 0, CF = 1, SF = 0, PF = 0 ו- OF = 0.

איך "משתמשים" בדגלים?

ה"שימוש" בדגלים, שתפקידם למעשה ליצג אינפורמציה על הפעולה האריתמטית האחרונה היא ע"י שימוש בפקודות הסתעפות מותנות. פקודות הסתעפות מותנות הן פקודות שיש להן שתי תכונות חשובות מאד:

1. קודם כל הן פקודות הסתעפות כלומר פקודות שמשנות את הפקודה הבאה לביצוע, כמו JMP.

2. בניגוד ל-JMP, ההסתעפות לא מתרחשת בכל מקרה. עבור כל פקודת הסתעפות מותנית, דגלי הבקרה חייבים לקיים תנאי מסוים בכדי שיתבצע הסתעפות. במידה והתנאי הזה אינו מתקיים, הפקודה הבאה לביצוע היא הפקודה העוקבת לפקודת ההסתעפות המותנית.

לדוגמא, ניקח את הפקודה JZ - "Jump on Zero Flag".
כמו בכל פקודת הסתעפות, היא מקבלת פרמטר יחיד - כמעט תמיד label.
כלומר בתוכנית היא תופיע בדרך כלל בצורה

```
JZ label
```

התנאי, שחיב להתקיים על מנת שתהיה הסתעפות, הוא ש- ZF = 1. ערך הדגלים האחרים אינם משפיעים על הפקודה הזו.

במידה ותוכנית מבצעת את פקודה JZ, ה-CPU בודק את ZF. במידה ו-ZF = 1, יתבצע הסתעפות לפקודה ש-label מצביע עליה - היא תהיה הפקודה הבאה לביצוע. במידה ו-ZF = 0, לא תתבצע הסתעפות - כלומר הפקודה הבאה לביצוע תהיה הפקודה העוקבת לפקודת ה-JZ. לדוגמא, נבחן את קטע הקוד הבא:

```
  SUB AX,BX
  JZ Alab1
  INC CX
Alab1:
  ADD AX,CX
```

בקטע הקוד הזה, הביצוע של הפקודה INC CX תלוי בערכים של AX ו-BX ברגע ביצוע הפקודה SUB AX,BX. במידה ותוכן AX ו-BX היו זהים, תוצאת החיסור שיוצב ב-AX יהיה אפס, והפקודה INC CX לא תתבצע.הפקודה JZ Alab1 תגרום לכך שנעקוף את הפקודה INC CX. במידה ותוכן AX ו-BX היו שונים, תוצאת החיסור שיוצב ב-AX יהיה שונה מאפס (חיובי או שלילי). הפקודה JZ Alab1 לא תשנה את הפקודה הבאה לביצוע. הוא ישאר הפקודה העוקבת, שהוא הפקודה INC CX, והפקודה INC CX תתבצע.

לכל אחד מהדגלים יש 2 פקודות הסתעפות מותנות התלויות רק בו:

```
JZ - "Jump on Zero (ZF = 1)"
JNZ  "Jump on Not Zero (ZF = 0)"

JC - "Jump on Carry (CF = 1)"
JNC  "Jump on Not Carry (CF = 0)"

JP - "Jump on Parity (PF = 1)"
JNP  "Jump on Not Parity (PF = 0)"
```

JS - "Jump on Sign (SF = 1)"
JNS "Jump on Not Sign (SF = 0)"

JO - "Jump on Overflow (OF = 1)"
JNO "Jump on Not Overflow (OF = 0)"

בשלב זה אפשר לסכם, שפקודות המכונה האריתמטיות קובעות את ערכי הדגלים ותוכניות המעונינות להתחשב בערכי הדגלים מסתעפים באופן מותנה לקטעי קוד רצויים בהתאם לערכי הדגלים. מה שלא ברור עדיין למה זה טוב. המטרה של המנגנון שתואר לעיל הוא לממש קוד המבוסס על תוצאות של השוואות מספרים. כלומר, מה שאנחנו רוצים לממש כאן הוא, במושגים של שפת C, אמצעי תכנות כמו

```
if (x == y)
  { ...
  }

while(x < y)
  { ...
  }
```

וכו'.

באסמבלי, קוד מהסוג הזה ממומש ע"י הפקודת המכונה CMP ופקודות הסתעפות מותנות. הפקודה CMP היא פקודה אריתמטית, ולמעשה היא פקודה הממומשת בצורה זהה ל-SUB, פקודת המכונה של החיסור, בהבדל אחד: בניגוד ל-SUB, תוצאת החיסור אינה נשמרת באופרנד היעד. בפקודה CMP, אופרנד היעד (כמו המקור) אינו משתנה. פעולת החיסור, משפיעה רק על הדגלים. לדוגמא,

```
        I                    II

    MOV  AX,6            MOV AX,6
    SUB  AX,2            CMP AX,2
```

בקוד I, אחרי פקודת ה-SUB הערך של AX הוא 4, ואילו בקוד II אחרי פקודת ה-CMP נשאר 6. אך תוכן אוגר הדגלים יהיה זהה בשני המקרים (ZF = 0, SF = 0 ...).

כאשר אנחנו משווים בין מספרים שלמים, נניח x ו-y, ישנם 2 אפשריות איך אנחנו מתיחסים למספרים: מספרים שלמים עם סימן, או מספרים שלמים אי שליליים (חסרי סימן).

יש גם אפשרויות שונות לגבי השימוש בתוצאה: יכול להיות שאנחנו מעונינים בתשובה לשאלה (במושגים של שפת C) "האם x == y", או "האם x < y", או "האם x <= y" וכו'.

בכדי להבין את המנגנון המממש את אמצעי התכנות שלעיל, יש להבין את משפט המפתח הבא:

"הפקודה CMP שומרת באוגר הדגלים את כל האינפורמציה הנחוצה לביצוע כל ההשוואות האפשריות בשתי צורות ההתיחסות למספרים השלמים".

הרעיון הוא שערכם של הדגלים CF, SF, ZF ו-OF מכילים את כל האינפורמציה הנחוצה לכל ההשוואות האפשריות בשתי צורות ההתיחסות למספרים. השימוש בדגלים אלו הוא ע"י פקודות הסתעפות מותנות, שההתניה שלהן הוא תנאי משולב על מספר דגלים.

פקודות ההסתעפות שבהם מדובר הם כלהלן:

<u>פקודות הסתעפות מותנות - מספרים חסרי סימן</u>:

JA - Jump if Above (JNBE - Jump if not Below or Equal)
JAE - Jump if Above or Equal (JNB - Jump if not Below)
JE - Jump if Equal (JZ - Jump if Zero)
JNE - Jump if Not Equal (JNZ - Jump if Not Zero)
JBE - Jump if Below or Equal (JNA - Jump if not Above)
JB - Jump if Below (JNAE - Jump if not Above or Equal))

<u>פקודות הסתעפות מותנות - מספרים עם סימן</u>:

JG - Jump if Greater (JNLE - Jump if not Less or Equal)
JGE - Jump if Greater or Equal (JNL - Jump if not Less)
JE - Jump if Equal (JZ - Jump if Zero)
JNE - Jump if Not Equal (JNZ - Jump if Not Zero)
JLE - Jump if Less or Equal (JNG - Jump if not Greater)
JL - Jump if Less (JNGE - Jump if not Greater or Equal))

במקרה של שיויון, פקודת ההסתעפות JE משותפת לשני סוגי המספרים עם או בלי סימן, משום שתוצאת אפס בחיסור זהה לשני המקרים - יש לנו אלגוריתם חיסור אחד בלבד. JE הוא זהה ל-JZ (Jump on ZF = 1). לנוחות המתכנת יש למרבית הפקודות ההסתעפות יותר משם אחד - JZ ו-JE, JA ו-JNBE, JGE ו-JNL הם שלושה מקרים של שתי שמות שיוצרים בדיוק אותה פקודת מכונה.

השימוש בפקודות אלו הוא כמעט תמיד בשילוב עם הפקודה CMP. על מנת להבין אותם נצא מתוך ההנחה שזה אכן המקרה.

נמחיש את הרעיון ע"י דוגמא:
נתון הקוד הבא:

```
MOV CX,1
CMP AX,BX
JA Skip1
  MOV CX,0
Skip1:
```

הקוד שלעיל יציב ל-CX את הערך 1 אם הערך של AX גדול ממש מהערך של BX כאשר שניהם מפורשים כמספרי חסרי סימן (אי שליליים). במידה ורצינו לקבל את אותו אפקט על מספרי עם סימן, ההבדל היחיד היה החלפת הפקודה JA ב-JG, כלומר הקוד יהיה:

```
MOV CX,1
CMP AX,BX
JG Skip1
  MOV CX,0
Skip1:
```

לדוגמא, אם AX ו-BX היו מכילים את ערכים חיוביים שהם פחות מ-32767, לא יהיה הבדל בביצוע של שתי גירסאות הקוד. לעומת זאת אם אחד משני האוגרים מקבל ערך חיובי 32768 או יותר ההתנהגות תהיה שונה, משום שהפרשנות של מספר כזה, כמספר עם סימן, נחשב למספר שלילי.

המימוש של הפקודות הללו, הוא על סמך הדגלים שציינו קודם:
למשל, הפקודה JA אפשר לכנות גם Jump if CF = 0 and ZF = 0.
הפקודה JG אפשר לכנות גם Jump if SF = OF and ZF = 0.
הפקודה JLE אפשר לכנות גם Jump if SF =/= OF or ZF = 1.
הפקודה JB זהה לפקודה JC (Jump if CF = 1).

וכו'. בדרך כלל אין לנו ענין לזכור בדיוק את הפרטים הטכניים הללו, אבל חשוב להבין שהדגלים מכלים את כל האינפומציה הנחוצה.

הגבלות

ב-8086, פקודות ההסתעפות המותנות, כלומר כל הפקודות ההסתעפות למעט JMP, מוגבלות לנקודה בתוכנית במרחק של לכל היותר 128- ... 127+ מפקודת ההסתעפות עצמה. פירוש הדבר שאם בתוכנית מופיעה הפקודה

```
JA label
```

אז label חייב להיות, קודם כל, באותו סגמנט שבו הפקודה הזו מופיעה.

בנוסף label חייב להיות בנקודת זיכרון (offset) שהוא לכל היותר 128 לפני הפקודה JA, או לכל היותר 127 אחרי הפקודה JA. בטרמינולוגיה של אינטל מדובר בקפיצה קצרה (Short Jump). חלק מהפקודה היא byte יחיד המכיל את השינוי (חיובי או שלילי) שיש להוסיף ל-offset של הפקודה הנוכחית כדי להגיע ליעד.

ואשר לפקודת ההסתעפות הבלתי מותנית JMP, ב-8086 היו לו שתי גירסאות:
אחת קצרה (כמו פקודת ההסתעפות המותנות) 128- ... 127+,
השניה ארוכה (Long Jump) 32768- ... 32767+.

כפי שיאמר בהמשך, המגבלה הזו של 128- ... 127+ לא קיימת תחת ה-386 (או תוכניות שבהם מופיע ההנחיה 386.). לכן המגבלה הזו לא צריכה להוות בעיה מיוחדת. יחד עם זאת, גם לתוכניות 8086 יש לזה פתרון.

במידה והיה צורך לממש פקודת הסתעפות למרחק גדול מ- 128- ... 127+, צריך לעקוף את הבעיה ע"י JMP. לדוגמא, אם אנחנו צריכים לממש

```
Lab:

  .....

  CMP AX,BX
  JE  Lab
```

ו-Lab נמצע רחוק מדי, ניתן להשתמש בפקודת ההסתעפות ההפוכה ל-JE (שהיא JNE) ולממש את הקוד בצורה

```
Lab:

  .....

  CMP AX,BX
  JNE  SkipJmpl  ;  תנאי הפוך!
  JMP  Lab
SkipJmpl:
```

לכל פקודת הסתעפות צריך לבחור את ההיפוך הלוגי שלו, למשל עבור JA זה JNA (או באופן שקול JBE), עבור JGE זה JNGE (או באופן שקול JL) וכו'.

תכנות מבני באסמבלי

המטרה של כל האמור לעיל, הוא מימוש תכנות מבני באסמבלי. נראה עכשיו

סידרה של דוגמאות של מבני תוכניות ב-C ואת מימושם באסמבלי. זה לא מכסה את כל האפשרויות אבל נעבור על מספיק קטגוריות בכדי שהדרך לממש תוכניות מהסוג הזה יהיה ברור.

בכל הדוגמאות ההנחה היא שהמגבלה של 128- ... 127- לא מהווה בעיה. במידה ויש בעיה כזו, ניתן לעקוף את הבעיה בצורה המתוארת שלעיל.

דוגמא מספר 1:

מימוש קוד מהסוג (במושגים של שפת C):

```
if (AX == BX)
 {
  .... גוף פקודות  ...
 }
```

באסמבלי:

```
CMP AX,BX
JNE Skip1   ; תנאי הפוך!
  .... גוף פקודות  ...
Skip1:
```

דוגמא מספר 2:

מימוש קוד מהסוג (במושגים של שפת C):

```
if (AX == BX)
 {
  .... 1 גוף פקודות  ...
 }
else
 {
  .... 2 גוף פקודות  ...
 }
```

באסמבלי:

```
CMP AX,BX
JNE Else1   ; תנאי הפוך!
  .... 1 גוף פקודות  ...
JMP EndIf1
Else1:
  .... 2 גוף פקודות  ...
EndIf1:
```

דוגמא מספר 3:

מימוש קוד מהסוג (במושגים של שפת C):

```
if ( (AX == BX) && (CX == DX )
 {
  .... גוף פקודות  ...
 }
```

באסמבלי:

```
  CMP AX,BX
  JNE SkipBody1   ; תנאי הפוך!
  CMP CX,DX
  JNE SkipBody1   ; תנאי הפוך!

  .... גוף פקודות ...

SkipBody1:
```

דוגמא מספר 4:

מימוש קוד מהסוג (במושגים של שפת C):

```
  if ( (AX == BX) || (CX == DX )
   {
    .... גוף פקודות ...
   }
```

באסמבלי:

```
  CMP AX,BX
  JE DoBody1    ; תנאי ישיר!
  CMP CX,DX
  JE DoBody1    ; תנאי ישיר!
  JMP SkipBody1
DoBody1:

  .... גוף פקודות ...

SkipBody1:
```

אפשר גם
JNE SkipBody
אבל רק לתנאי לאחרון

דוגמא מספר 5:

מימוש קוד מהסוג (במושגים של שפת C):

```
  while (AX == BX)
   {
    .... גוף פקודות ...
   }
```

באסמבלי:
פתרון אינטואיטיבי:

```
DoBody1:
  CMP AX,BX
  JNE Fin1    ; תנאי הפוך!

  .... גוף פקודות ...

  JMP DoBody1
Fin1:
```

אפשרות אחרת (יעילה מעט יותר):

```
 JMP TestNext1
DoBody1:

  .... גוף פקודות ...

TestNext1:
  CMP AX,BX
  JE  DoBody1   ; תנאי ישיר!
```

דוגמא מספר 6:

מימוש קוד מהסוג (במושגים של שפת C):

```
  do
   {
    .... גוף פקודות ...
   } while (AX == BX);
```

באסמבלי:

```
DoBody1:

  .... גוף פקודות ...

  CMP AX,BX
  JE  DoBody1   ; תנאי ישיר!
```

דוגמא מספר 7:

מימוש קוד מהסוג (במושגים של שפת C):

```
  for(DX = 0; DX < N; DX++)
   {
    .... גוף פקודות ...
   }
```

באסמבלי:

פתרון אינטואיטיבי:

```
  MOV DX,0
 For1:
  CMP DX,N
  JNL Fin1

    .... גוף פקודות ...

  INC DX
  JMP For1
 Fin1:
```

פתרון יעיל מעט יותר:

```
      MOV  DX,0
      JMP NextCheck1
    DoBody1:

      .... גוף פקודות ...

      INC DX
    NextCheck1:
      CMP DX,N
      JL  DoBody1   ; תנאי ישיר!
```

ההנחה כאן ש-DX ו-N מפורשים כמספרים עם סימן, מכאן השימוש ב-JGE ו-JL. אחרת (התיחסות אליהם כמספרים חסרי סימן) יש להחליפם ב-JAE ו-JB.

הפקודה LOOP

ראינו שניתן לממש את אמצעי התכנות המבוסס על מניה (for במושגים של C ומרבית שפות העילית האחרות) ע"י CMP והסתעפיות מותנות. זוהי הצורה ש-TURBO C מממש את המבנה הזה. עם זאת, ישנה תמיכה בחומרה למימוש יותר יעיל (לפחות ב-8086) של המבנה הזה.

אפשר לומר שיש תמיכה באופן ספציפי למבנה הבא (במושגי שפת C):

```
for(CX = N; CX != 0; CX--)
{
  .... גוף פקודות ...
}
```

התמיכה בחומרה למבנה הזה הוא בראש ובראשונה הפקודה LOOP שהשימוש שלה הוא מהצורה

```
LOOP label
```

מה שהפקודה עושה הוא כלהלן:

1. מבצע הפחתה של CX (מעין DEC CX אוטומטי).
2. במידה ו- CX > 0, מתקים הסתעפות ל-label (הסתעפות מותנית במקרה של 0 > CX).

LOOP היא איפוא פקודה אחת שממשת את הקוד הבא:

```
      DEC CX
      JNZ label
```

החסכון מתקבל מכך שאין צורך לקרוא שתי פקודות מהזיכרון ולפענח אותם (מעבר לזמן הנחוץ לבצע אותם). אגב הבדל נוסף הוא ש-LOOP אינו מבצע את הבדיקה באמצעות הדגלים ואינו משפיע עליהם.

ישנה גם פקודת הסתעפות מותנית JCXZ - Jump if CX is Zero המבצעת הסתעפות

מותנית אם CX = 0, שוב ללא שימוש בדגלים וללא השפעה עליהם.
גם LOOP וגם JCXZ חייבים לקיים את ההגבלה של הסתעפיות קצרות (128- ... +127).

לפיכך המימוש של ה-for האחרון (המונה כלפי אפס) באסמבלי הוא כלהלן:

```
  MOV CX,N
  JCXZ SkipLoop1
Do1:
  .... גוף פקודות ...
  LOOP Do1
SkipLoop1:
```

לפקודה LOOP יש גירסאות LOOPE (נקרא גם LOOPZ) ו-LOOPNE (נקרא גם LOOPNZ) שמתנות את ההסתעפות בכך ש-ZF = 1 (ZF = 0) בנוסף לתנאי ש- CX > 0.
לדוגמא הלולאה הבאה תסתים כאשר CX = 0 או ש- AX = BX ברגע ביצוע פקודת ה-LOOP:

```
  MOV CX,N
  JCXZ SkipLoop1
Do1:
  .... גוף פקודות ...
  CMP AX,BX
  LOOPNE Do1
SkipLoop1:
```

<u>מעבדי 386 ואילך</u>

אוגר הדגלים של המעבדים 386 אמנם הורחב ל-32 ביט, אבל הדגלים הנוספים אינם קשורים לנושאים שנדונו בתקציר זה. לפיכך אין הבדל בין ה-8086 ל-386 בהקשר הזה.

ב-386 הפקודה CMP והפקודות האריתמטיות יכולות לפעול על אופרנדים 32 ביט (EAX, EBX, ...) וההשפעה על הדגלים זהה לגירסאות 16 ביט.

אולי ההבדל החשוב ביותר בין ה-386 ל-8086 בהקשר של התקציר הזה הוא ביטול ההגבלה של 128- .... 127+ על פקודות ההסתעפות המותנות (JE, JNE, JC,...). תחת 386. ההסתעפות יכולה להיות 16 ביט (32768- ... 32767+) ובתנאים מסוימים אפילו 32 ביט.

הבדל נוסף הוא שב-386 הפקודה JMP יכול להסתעף לסגמנט אחר (כלומר שינוי CS בנוסף ל-EIP). רק JMP יכול לבצע את ההסתעפות הבין סגמנטית הזאת.

אשר לפקודות ה- LOOP, LOOPE, LOOPNE, ב-386 ישנן לפקודה הללו גירסאות שבהם אוגר הבקרה הוא ECX וההסתעפות אפשרית למרחקים 32768- עד 32767+. הפקודות הללו נקראות LOOPD, LOOPDE, LOOPDNE בהתאמה.

לא ודאי שכדאי להשתמש בגירסאות של הפקודה LOOP במעבדים הללו: יתכן שהשילוב

```
DEC CX
JNZ label
```

או

```
DEC ECX
JNZ label
```

הוא יותר מהיר.

במקביל לפקודה JCXZ יש ב-386 פקודה נוספת JECXZ - Jump if ECX is Zero.

<u>דוגמא מסכמת</u>

להלן תוכנית המממשת את האלגוריתם של אוקלידס באסמבלי, בצד המימוש שלו ב-C. נקודה מענינת כאן היא שבאסמבלי, אפשר "לחסוך" השוואות: ניתן לבצע יותר מבדיקה אחת עם התוצאות של פקודת CMP אחת.

```
;
; gcd3.asm - Compute greatest common divisor, 386 version
;
    .MODEL SMALL
    .STACK 100h
    .DATA
x    DD  881790
y    DD  188955
Gcd  DD  ?
     .CODE
     .386            ;  Enable 386 code
     MOV AX,@DATA    ; Program prefix
     MOV DS,AX       ; Set DS to point to data segment
                     ;
     MOV EAX,x       ;  First operand
     MOV EBX,y       ;  Second operand          ***** C *****
Do1:                 ;                           while (eax != ebx)
     CMP EAX,EBX     ;                             {
     JE  Endlp       ; Exit if EAX = EBX,                if (eax<ebx)
       JAE Xhigh     ; Skip XCHG IF EAX=>EBX               {
         XCHG EAX,EBX ; EAX < EBX, Swap them               temp = eax;
                     ;                                     eax = ebx;
                     ;                                     ebx = temp;
       Xhigh:        ;                                   }
         SUB EAX,EBX ; EAX := EAX - EBX                  eax = eax - ebx;
     JMP Do1         ;
Endlp:               ;                             } /* while */
     MOV Gcd,EAX     ; Store result                gcd = eax;
                     ;
     MOV AH,4Ch      ; Set terminate option for int 21h
     INT 21h         ; Return to DOS (terminate program)
     END
```