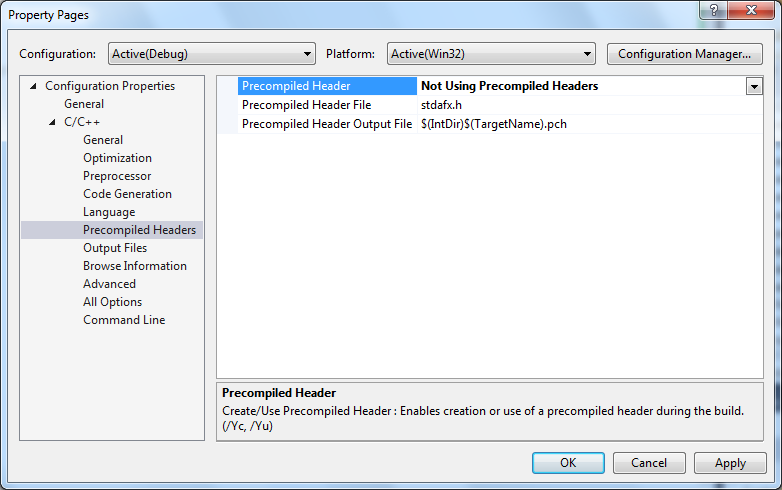


C topics that helps the assembler project

שלב ראשון, עדיף לעבוד עם ה visual studio

לשנות את כל הקבצים ל סיומת C למחוק את stdafx.h



Bit field, and especially a union for 32 bit or 16 bit for bit field , will help use to access bits instead of doing shift , and other operations to extract bits

union cpuOpcode

{

struct

{

unsigned int era : 2;

unsigned int opDest : 2;

unsigned int opSource : 2;

unsigned int opcode : 4;

unsigned int group : 2;

unsigned int unused : 3;

} u;

unsigned int fullReg;

};

For example:

cpuOpcode r;

r.u.era = 1;

r.u.opcode = 3;

int x = r.fullReg;

A 16 bit register will be:

union cpuOpcode

{

struct

{

unsigned short era : 2;

unsigned short opDest : 2;

unsigned short opSource : 2;

unsigned short opcode : 4;

unsigned short group : 2;

unsigned short unused : 3;

} u;

unsigned short fullReg;

};

[token concatenation](http://en.wikipedia.org/wiki/C_preprocessor#Token_concatenation).

<http://stackoverflow.com/questions/3318091/generating-labels-in-macro-preprocessing-in-c>

נרצה להשתמש ביכולת הזו בשביל פקודות JUMP ו LABLE

#define JUMP(f) goto f##\_LABLE

#define LABEL(x) x##\_LABLE:

void foo(void)

{

printf("At the beginning.\n");

JUMP(here);

printf("You shouldn't see this.\n");

LABEL(here);

printf("At the end.\n");

return;

}

אני רוצה להראות שיטה מסויימת לכתוב תוכניות דומות לאלה , זה גם מאד שימושי בעתיד שיבוא במקרה ועובדים בחברות של שבבים, ורוצים לבנות סימולטור לרכיב של החברה.

בתוכנית שצריך לבנות יש כמה שלבים

1. שלב של קליטת התוכנית ובדיקת syntax
2. אחרי שקלטנו את התוכנית צריך לבצע מעבר ראשון ולייצר אותו
3. מעבר שני , החלפת ה label בכתובות , מיעון ישיר ועוד
4. הרצת התוכנית, על ידי קוד בינארי

נניח שאנחנו לא רק לבצע רק את שלב 2 ו 3 אבל גם להריץ את התוכנית

אנחנו יכולים לעשות כן על ידי כך שהתוכנית שלנו תהיה תוכנית שנקמפל ונריץ

דרך זו תעזור לנו בשלב ראשון לבדוק את האסמבלר שלנו בלי הרבה עבודות string

לפני שנתחיל במימוש פקודות , נגדיר כמה דברים בסיסיים:

בקובץ header:

typedef union registers

{

struct

{

short r : 15;

short unused : 1;

} u;

short fullReg;

} registers;

typedef union status\_register

{

struct

{

short c : 1;

short z : 1;

short unused : 14;

} u;

short fullReg;

} status\_register;

typedef union cpu\_memory

{

struct

{

short mem : 15;

short unsued : 1;

} u;

short fullReg;

} cpu\_memory;

registers cpu\_regs[7];

cpu\_memory cpu\_mem[1000];

cpu\_code[10000];

typedef enum CPU\_REGISTERS

{

r0,

r1,

r2,

r3,

r4,

r5,

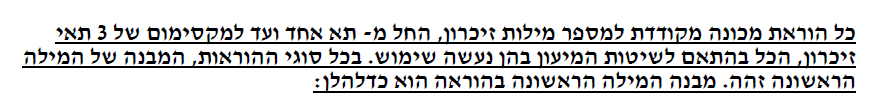
r6,

r7

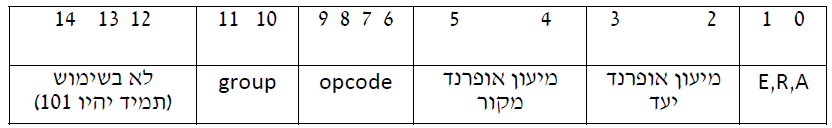
} CPU\_REGISTERS;

מכיוון שהמיעונים הם מיעוני אופרנד זה כבר אומר שיהיו עוד שורות בזכרון לרבות השורה הראשונה

וזה בדיוק מה שאומרים לנו במאמר מודגש:



זוהי השורה הראשונה בזכרון הקוד של התוכנית והיא תמיד תהיה.



עבורה הגדרנו:

typedef union cpuOpcode

{

struct

{

unsigned short era : 2;

unsigned short opDest : 2;

unsigned short opSource : 2;

unsigned short opcode : 4;

unsigned short group : 2;

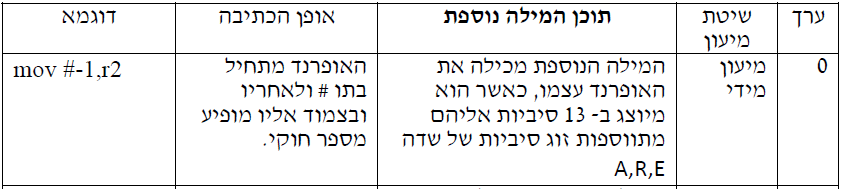
unsigned short unused : 3;

} u;

unsigned short fullReg;

} cpuOpcode;

נתחיל מפקודה ראשונה mov\_immediate



הפקודה הזו לא קיימת אני המצאתי אותו על מנת קודם לעבוד על האסמבלר.

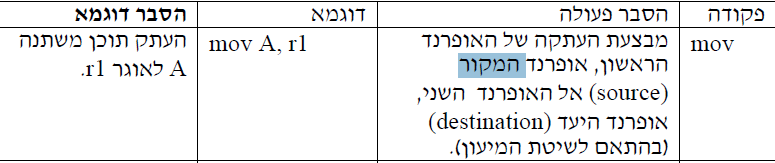
שנסיים עם כל הפקודות נבנה parser שיעבוד רק עם פקודה אחת mov

מבנה הפקודה שלנו הוא:

void mov\_immediate(int source\_operand, CPU\_REGISTERS dest\_reg)

איך יודעים מי ה source ומי ה destination בפקודה?

רשום למטה:



עבור המילה השנייה הגדרתי typedef נוסף שיהיה לי קל להציב ביטים איתו:

typedef union cpuOpcode2

{

struct

{

unsigned short era : 2;

unsigned short direct\_operand : 13;

unsigned short unused : 1;

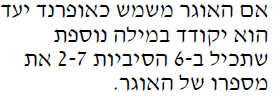
} u;

unsigned short fullReg;

} cpuOpcode2;

האופרנד השני בדוגמה זו הוא אופרנד אוגר , ורשום לנו שזה מה שצריך לעשות שהאופרנד הוא אוגר

רק להזכיר , אנחנו חייבים להוסיף מילים מכיוון שהמיעונים הם רק 2 ביטים ויש לנו 8 אוגרים ו זכרון של 13 ביט



typedef union cpuOperandDestination

{

struct

{

unsigned short era : 2;

unsigned short reg\_num : 6;

unsigned short unused : 8;

} u;

unsigned short fullReg;

} cpuOperandDestination;

הקוד הסופי שלנו למימוש פקודת ה mov immiediate הוא:

שימו לב שחשוב לאפס את fullreg

// Example move -1, r2

void mov\_immediate(int source\_operand, CPU\_REGISTERS dest\_reg)

{

cpuOpcode o;

o.fullReg = 0;

o.u.era = ARE\_ABSOLUTE;

o.u.opSource = IMMEDIATE\_ACCESSING;

o.u.opDest = REGISTER\_DIRECT\_ACCESSING;

o.u.opcode = OPCODE\_MOV;

o.u.unused = 0x5;

o.u.group = TWO\_OPERANDS;

pushCommand(o.fullReg);

cpuOpcode2 m2;

m2.fullReg = 0;

m2.u.era = ARE\_EXTERNAL;

m2.u.direct\_operand = source\_operand;

pushCommand(m2.fullReg);

cpuOperandDestination m3;

m3.fullReg = 0;

m3.u.era = ARE\_ABSOLUTE;

m3.u.reg\_num = dest\_reg;

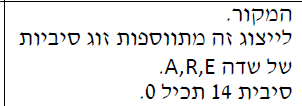
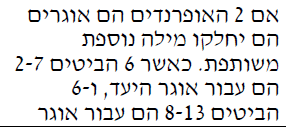
pushCommand(m3.fullReg);

}

כרגע לא נוכל לבדוק אם הוא נכון כי אין לנו דוגמא שנתנו לנו , נבדוק בהמשך

מימוש פקודת sub r1,r2

פקודה זו היא פקודת מיעון אוגר ב מקור וביעד , יש לנו דוגמא בסוף החוברת איך היא צריכה להיות בסוף.

נבנה לזה typedef נוסף

typedef union cpuOperandSourceAndDestination

{

struct

{

unsigned short era : 2;

unsigned short dest\_reg\_num : 6;

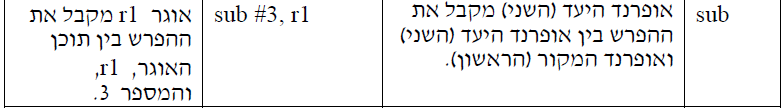
unsigned short source\_reg\_num : 6;

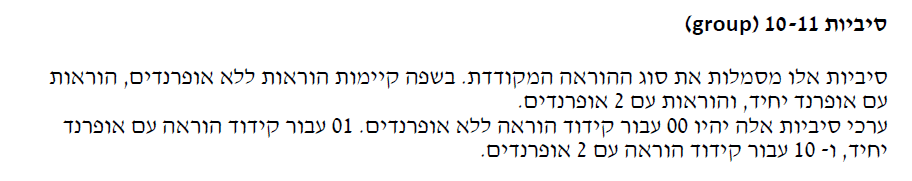
unsigned short unused : 1;

} u;

unsigned short fullReg;

} cpuOperandSourceAndDestination;





הפקודה sub היא שני אופרנדים

לכן הgroup שלה יהיה 0x2

הקוד הסופי לפקודת ה Sub היא:

// Example sub\_registers r1,r4

void sub\_registers(CPU\_REGISTERS source\_reg, CPU\_REGISTERS dest\_reg)

{

cpuOpcode o;

o.fullReg = 0;

o.u.era = ARE\_ABSOLUTE;

o.u.opSource = REGISTER\_DIRECT\_ACCESSING;

o.u.opDest = REGISTER\_DIRECT\_ACCESSING;

o.u.opcode = OPCODE\_SUB;

o.u.group = TWO\_OPERANDS;

o.u.unused = 0x5;

pushCommand(o.fullReg, 0);

cpuOperandSourceAndDestination m2;

m2.fullReg = 0;

m2.u.era = ARE\_ABSOLUTE;

m2.u.source\_reg\_num = source\_reg;

m2.u.dest\_reg\_num = dest\_reg;

m2.u.unused = 0;

pushCommand(m2.fullReg, 1);

}

בפקודה הזו אמורים להיות שתי שורות או מילים של 15 ביט כול אחת

נבנה תוכנית main עם הפקודה הזו בלבד:



int \_tmain(int argc, \_TCHAR\* argv[])

{

sub(r1, r4);

return 0;

}

התוצאה שהתקבלה:

101-10-0011-11-11-00

000000100010000

והיא זהה לתוצאה שיש בדוגמא:



זה מה שהתכוונתי שאנחנו בונים את התוכנית לפני שלבים.

מימוש פקודת INC K

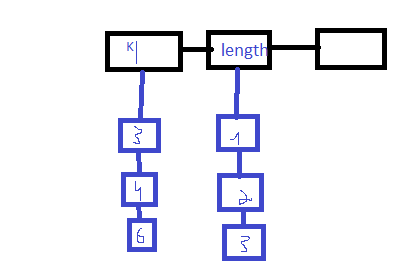
בשביל להתחיל להבין את הפקודה , צריך להבין מהיכן הגיע K

השפה יכולה להגדיר data image עם ערכים מופרדים ב פסיקים

בתוכנית לדוגמא מופיע K: .data 2

LENGTH: .data 6,-9,15

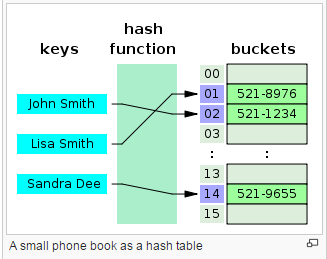
בהמשך נצטרך לקרוא את השורה הזה מקובץ ה text ולהכניסו למבנה נתונים גמיש שנוכל לגשת אליו



המבנה הזה יכול להספיק , אולם אם מספר המשתנים יהיה ענק אז החיפוש יהיה O(n) אם לא ניישם hash table with has function

In computing, a **hash table** (**hash** map) is a data structure used to implement an associative array, a structure that can map keys to values. A **hash table** uses a **hash** function to compute an index into an array of buckets or slots, from which the desired value can be found.

זה בדיוק מה שאנחנו צריכים בשביל לשמור את ה keys של תויות ה data



<https://www.youtube.com/watch?v=h2d9b_nEzoA>

צריך לבחור good hash function that will limit the collision as much as possible

Once there are collisions, a linked list should be implemented to add the values into the same bucket

int hash(const char\* word)

{

unsigned int hash = 0;

for (int i = 0; word[i] != '\0'; i++)

{

hash = 31 \* hash + word[i];

}

return hash % 1000;

}

An example of hash function with a range of 1000 (because of the modulo)

There are many hash function out there to use before inventing one.

בהמשך נרחיב על כל ה hash כולל ישום, בינתיים נראה קצת highlights

כך נראה ה bucket list of keys

node \*\* buckets = (node \*\*)malloc(20 \* sizeof(node));

node \*\*p[2];

p[0] = buckets;

unsigned long hcode;

hcode = hash("This is a string", 20);

buckets[hcode] = (node \*)malloc(sizeof(node));

strcpy(buckets[hcode]->str, "This is a string");

buckets[hcode]->next = NULL;

hcode = hash("This is a 2 string", 20);

buckets[hcode] = (node \*)malloc(sizeof(node));

strcpy(buckets[hcode]->str, "This is a 2 string");

buckets[hcode]->next = NULL;

hcode = hash("This is a 3 string", 20);

buckets[hcode] = (node \*)malloc(sizeof(node));

strcpy(buckets[hcode]->str, "This is a 3 string");

buckets[hcode]->next = NULL;

מה קורה עם רוצים עוד keys?

הפעולה נקראת rehasing

היא פעולה יקרה בזמן, כדי להמנע ממנה ככל האפשר.

בפעולה זו אנחנו מגדירים hash חדש עם יותר מקומות ומעתיקים ובונים את ה hash table מחדש.

פעולת ה rehash תעשה גם 50% לפני הסוף מכיון שלא רוצים הרבה colision and chaining

במעבר הראשון אנחנו לא יודעים את הכתובת של K וקיבלנו את הערכים הבאים לפקודה:



101-01-0111-00-11-00

000000000000010 -- נתעלם מזו הפעם

הפונקציה של , INC יהיה לה מעבר נוסף לקביעת הכתובת של K

void inc\_direct(char address)

{

cpuOpcode o;

o.fullReg = 0;

o.u.era = ARE\_ABSOLUTE;

o.u.opSource = 0;

o.u.opDest = REGISTER\_DIRECT\_ACCESSING;

o.u.opcode = OPCODE\_INC;

o.u.group = ONE\_OPERANDS;

o.u.unused = 0x5;

pushCommand(o.fullReg, 0);

cpuOperandDestination m2;

m2.fullReg = 0;

m2.u.era = ARE\_RELOCATABLE;

m2.u.reg\_num = 0; // Address is not matter in the first pass

m2.u.unused = 0;

pushCommand(m2.fullReg, 1);

}

פקודות מיעון מיידי דינמי , לדוגמה:

mov -4[5-9],r3

נגדיר פונקציה זמנית בשם זה:

void mov\_imm\_dynamic(char \*address, CPU\_REGISTERS reg)

נרשום את הערכים של הפקודה של המילה הראשונה

cpuOpcode o;

o.fullReg = 0;

o.u.era = ARE\_ABSOLUTE;

o.u.opSource = REGISTER\_DIRECT\_ACCESSING;

o.u.opDest = REGISTER\_DIRECT\_ACCESSING;

o.u.opcode = OPCODE\_MOV;

o.u.group = TWO\_OPERANDS;

o.u.unused = 0x5;

pushCommand(o.fullReg, 0);

הדפסה למסך:

void putDash(int type, int count)

{

if (type == 0)

{

if (count == 2)

printf("-");

else

if (count == 4)

printf("-");

else

if (count == 8)

printf("-");

else

if (count == 10)

printf("-");

else

if (count == 12)

printf("-");

}

}

פונקציה רקורסיבית על מנת להדפיס הפוך

int printb(int n, int i, int type)

{

int count = 0;

if (i == 0)

return 0;

count = printb(n >> 1, i - 1, type);

if (n & 1)

printf("1");

else

printf("0");

n >>= 1;

putDash(type, count);

count++;

}

void pushCommand(unsigned short cmd, int type)

{

printb(cmd, 15, type);

printf("\n");

}