Rasm51E Cross Assembler for the 8051 Microcontroller
User's Manual

6.115 Microcomputer Laboratory
Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

Introduction

Rasm51E is a cross assembler for the Intel MCS-51 assembly language used by the 8031/8051 family of microcontrollers. A cross assembler is a piece of software that runs on one computer and produces code for another. In 6.115, we will use IBM-PC style computers running Windows to run Rasm51E. We use an editor (notepad, emacs, etc.) on the PC to write code in assembly language for the 8051 microcontroller. Rasm51E produces a lower level Intel hex file. We use Hyperterminal to download this hex file to the R31JP microcontroller development board that we will use in 6.115.

Rasm51E is a two-pass assembler. Forward references are resolved during the second pass.

Using Rasm51e on the PC

In 6.115, we will be programming our R31JP microcontroller board in assembly language. You can write your program in any text editor (Emacs, vi, edit, Notepad, etc.), but be sure to save it as a ".asm" file.

On the 6.115 laboratory computers, you will run Rasm51E in a DOS window. We have written a batch file for you that calls Rasm51E with a few useful command line parameters. You will see this batch file demonstrated during your laboratory familiarization. The batch file is called RASM and includes a flag to generate a list (lst) file. The list file can be useful to you when debugging a program, please take the time to look at this file for a code example, and also look at the other files Rasm51E can produce to help you. The description below explains the command line options that you can use with Rasm51E if you want different behavior or information from that provided by our RASM batch file.

Rasm51E has a few options activated by the command line parameters.
Invoke Rasm51E from DOS by typing

Rasm51E filename[.asm] [-c] [-d|-dx] [-h|-h1|-h2] [-l|-lx] [-o] [-s]

Note that the command line parameters as well as the asm extension are optional. The command line parameters are described below.

COMMAND LINE OPTIONS

-c case sensitive labels

The c option causes the assembler to save the labels and symbols in a case sensitive fashion. Thus Label1 and label1, for example, are interpreted as two different labels.

-d display assembly progress

The d option displays the progress of the assembler on the standard output device, which by default is the screen. The output may be redirected to another device or file. For example,

Rasm51E myprog.asm -d > myprog.out

redirects the progress report to the file myprog.out. Similarly, the output may be redirected to a port.

Rasm51E myprog.asm -d > prn

redirects the progress report to the standard print device connected to LPT1 (parallel port 1), and

Rasm51E myprog.asm -d > lpt2

redirects the progress report to the second parallel port, LPT2. See your DOS manual for further information on output redirection.

-dx extended display assembly progress

This option is similar to the d option. It gives further information about the assembly process. For example, it notifies if a label was resolved during the first pass or during the second pass. See option d to redirect the output.

-h1 hold (terminate) on pass 1 errors

When used, the -h1 option prevent the assembler from proceeding with the second pass. Note that Pass 1 errors does not include forward references not resolved during the first pass. Unrecognized operations, operands or pseudo operations, illegal labels, and fatal errors (such as memory allocation or file creation errors) are reported during the first pass. This is a useful option to check a long source code for possible errors before generating a list file.

-h or -h2 hold (terminate) on after pass 2 errors

When used, the -h (or -h2) option prevent the assembler from creating a LST or OBJ file.

-l write regular lst file

This option causes the creation of a LST file. The LST file will have the same name as the source (assembly) file, and the extension lst. If the file already exists, it will be overwritten. Any include files specified in the source code will not be listed in the LST file.

-lx write extended lst file (descends into include files)

This option is similar to the -l option. Any include files specified in the source code will be listed in the LST file, that is, the LST file will descend into the include files..

-o write obj file (in Intel Hex format)

This option causes the creation of a OBJ file in the Intel hex format. The OBJ file will have the same name as the source (assembly) file, and the extension obj. If the file already exists, it will be overwritten.

-s generate symbol list (appended to lst file)

When invoked, this option will generate a list of labels and symbols specified in the source (assembly) file. The label and symbol lists will be appended to the LST file.

PSEUDO OPERATIONS

There are three pseudo-operations recognized by Rasm51E. These are db (define byte), dw (define word), and org (origin) pseudo-operations.

COMPILER DIRECTIVES

The only compiler directive is the include directive used as

#include filename.ext

where filename.ext must be found in the current directory or path. The file filename.ext will be opened and merged with the source (assembly) code. The number of include files is limited by the DOS parameters BUFFERS and FILES. See your DOS manual for further details. Up to 8 include file may be nested. That is, include files may be specified inside include files, stacked up to 8 levels.

EXPRESSIONS

The four basic arithmetic operations (+ - * /) are supported. Parentheses may be used to group terms of an expression. The Parentheses may be nested. The number of such nestings is limited only be the amount of dynamic memory available.

FUNCTIONS

There are two built-in functions: high() and low(). They return the high byte and the low byte of a word (two-byte expression), respectively.

CONSTANTS

Constants may be decimal, binary, octal, hexadecimal, or ASCII. Hexadecimal constants must start with a numerical digit between 0 and 9. The suffixes b, o, d, and h are used to denote binary, octal, decimal, and hexadecimal constants, respectively. If no suffix is present, the

constant is assumed to be decimal. ASCII constants are placed within single quotation marks. See the following examples.

10, 10d, 10D, 0ah, 0aH, 0Ah, 0AH, 12o, 12O, 1010b, 1010B all have the value 10.
'A' has the value 65 or 41h.

EXAMPLE

Run Rasm51E with the example code DEMO.ASM given below. Try the -l and -lx options and observe the LST files generated in each case. Also try the -d and -dx options. DEMO.ASM will not assemble with the -c option, since some symbols are written without regard to cases, e.g. "One" and "one".

DEMO.ASM

```
value equ 6/2*3*2/3

     mov a, value
     mov a, high(value*100h)
     mov b, low(value*100h)
     mov a, #high(value*100h)
     mov b, #low(value*100h)
l1:
     mov a, (five/two)-
one+two+three+four+(((((five)+five)+five)+five)+five)

     anl a, One
l2:
     orl a, oNe
l3:
     org 8000h
l4:
     orl b, #ONE

     org 0a00h

#include demo1.inc
; end of file
```

DEMO1.INC

```
; first level include file

  one equ 1
```

```
  two equ (one+one)
  three equ (one+two)

#include demo2.inc



--------------------------------------------------------------------
DEMO2.INC
--------------------------------------------------------------------

; second level include file


 four equ (two+two)
 five equ (two+three);
```

RASM Batchfile

To assemble a program on the 6.115 lab computers with the RASM batch file, open a command prompt, change into the directory containing your .asm file, and type:

```
rasm filename.asm
```

RASM will create the Intel HEX format object file (filename.obj) in the current directory. This is the file that you will download to your R31JP with Hyperterminal.

TIPS

There are several things that you should keep in mind when writing your programs. If you're having trouble assembling your code, i.e., if RASM is reporting errors or not completing assembly, then you should double-check your code. Some problems to look for include:

File names

Filenames should be limited to no more than 8 characters.

Numbers

Make sure that you understand what you are trying to do when working with numbers in assembly language. When you use a number, such as the hexadecimal number A0 (letter "A" followed by a "zero"), RASM/Rasm51E will interpret the number based on the context of the

instruction you are using. Consider, for example, the following six lines of code, all of which are an attempt to move the contents of Port 2 into the accumulator:

```
mov A, 0x0A0h
mov A, 0A0h
mov A, P2
mov acc, 0A0h
mov acc, P2
mov A, A0h
```

The sixth line (**in bold**), mov A, A0h, will generate an error when assembled by Rasm51e. Numbers for the assembler must begin with a number, not a letter. If the assembler finds something that you think is a number, but that in fact begins with a letter, it will interpret the "number" as a text word, and become "confused." Make it clear to the assembler that you mean a number by beginning the number with a digit, even for hex numbers. For example, 0A0h (where the h specifies hex) will be fine.

Here's another example, an attempt to put a number directly into the accumulator. If you wanted the actual number FF to be in the accumulator, you would use the following command:

```
mov a, #0FFh
```

Remember that when you want to use a number, always put a digit at the beginning of it. For example, if you tried to perform the previous operation by writing

```
mov a, #FFh
```

you would receive an error message from the assembler.

A versus ACC

The accumulator is a commonly used register in assembly language programs. Sometimes, we refer to the accumulator with A, for example:

```
mov A, P2
```

On the other hand, we can also refer to the accumulator with ACC. For example, the same result can be achieved with the line of code:

```
mov acc, P2
```

Even though these two lines are legitimate, and will assemble, and will achieve the same result, they are not the same assembly language instruction! The command "mov A," is an actual single op-code in the 8051 assembly language. We can use the shorthand "A" to refer to the accumulator in this case because it is part of a specific op-code. The flag "acc" is shorthand

recognized by the assembler, which causes the assembler to substitute the actual hex numbered address for the accumulator (0E0h) into the code during assembly. Using “acc” results in the use of a different “mov” op-code than does the use of “A”. The use of “A” in this case will result in smaller code (explain why).

Some commands that we might like to use do not have an “A” version. For example, it is not possible to use the command:

push A

to push the value of the accumulator on the stack. This op-code does not exist, and the line of code will assemble with an error! It is perfectly fine to use this line of code, however:

push acc

Using “mov”

Make sure that when you write your programs, you use movx or movc when appropriate. Consult your 8051 manual if you are in doubt as to which one you should use. This can make errors in your program that prevent it from working properly, but rasm will not always catch it.

It is always good to check that no errors occurred during assembly. Sometimes an error message will appear, but at the end rasm will still say “no errors found.” This can cause you to not notice errors and make other changes that do not fix the problem. Always check the lines above the "no errors found" line.

Labels and Reserved Words

When using a jump or a call, it is important that you do not use certain reserved words as the name of the subroutine you are referencing. Jumps and calls can take constant values, so if you make a forward reference to a label with the same name as a predefined constant value, then the constant value will be substituted in assembler pass 1 rather than the usual substitution in pass 2. Therefore, in addition to any constants you've defined for yourself, there are a number of label names that you should avoid. These reserved words are listed below. One easy way to do this would be to prefix all of your labels with a letter like L.

acc, acc.0, acc.1, acc.2, acc.3, acc.4, acc.5, acc.6, acc.7,
b, b.0, b.1, b.2, b.3, b.4, b.5, b.6, b.7,
cprl2, ct2, cy,
dph, dpl,
ea, es, et0, et1, ex0, ex1, exen2, exf2, exti0, exti1,
f0,
ie, ie.0, ie.1, ie.2, ie.3, ie.4, ie.5, ie.6, ie.7, ie0, ie1,
int0, int1, ip, ip.0, ip.1, ip.2, ip.3, ip.4, ip.5, ip.6, ip.7,

it0, it1,
ov,
p, p0, p0.0, p0.1, p0.2, p0.3, p0.4, p0.5, p0.6, p0.7,
p1, p1.0, p1.1, p1.2, p1.3, p1.4, p1.5, p1.6, p1.7,
p2, p2.0, p2.1, p2.2, p2.3, p2.4, p2.5, p2.6, p2.7,
p3, p3.0, p3.1, p3.2, p3.3, p3.4, p3.5, p3.6, p3.7,
pcon, ps,
psw, psw.0, psw.1, psw.2, psw.3, psw.4, psw.5, psw.6, psw.7,
pt0, pt1, px0, px1, rb8,
rcap2h, rcap2l, rclk, rd, ren, reset, ri, sbuf,
scon, scon.0, scon.1, scon.2, scon.3, scon.4, scon.5, scon.6, scon.7,
sint,
sm0, sm1, sm2, sp,
t0, t1,
t2con, t2con.0, t2con.1, t2con.2, t2con.3, t2con.4, t2con.5, t2con.6, t2con.7,
tb8, tclk,
tcon, tcon.0, tcon.1, tcon.2, tcon.3, tcon.4, tcon.5, tcon.6, tcon.7,
tf0, tf1, tf2, th0, th1, th2, ti, timer0, timer1, tl0, tl1, tl2, tmod, tr0, tr1, tr2, txd,
wr

The EQU Directive

All EQU definitions must occur near the top of the assembly file, **before** you use the definition. Constants defined with *equ* should not start with *R* or *r*. When such constants are used by inc or dec instructions, for example, RASM incorrectly reserves three bytes for the instruction even though it only takes two. The third byte does not get filled and will contain unknown data when you download it.

```
;; BAD: assembles to 15 55 ?? 00
        ref  equ 55h
        dec ref
        nop


 ;; OK: assembles to 15 55 00
        xref equ 55h
        dec xref
        nop
```