

ANADOLU UNIVERSITY
DEPT. OF ELECTRICAL & ELECTRONICS ENGINEERING
EEM489 MICROPROCESSORS II LABORATORY
LAB 7: INTRO. TO GNU C COMPILER FOR M68K

BEFORE LAB: READ THIS SHEET CAREFULLY
AND GATHER NECESSARY INFORMATION

Jan 4th, 2010
TA: ŞÜKRÜ GÖRGÜLÜ

1. Installation of GNU C Cross Compiler

GCC is a general and open source set of embedded software libraries in C, C++ and Assembler for robotics programming, targeting controllers using Motorola 68K/ColdFire & PowerPC micros. You can download its various OS-compatible versions from the generic following link:

http://sourceforge.net/project/showfiles.php?group_id=39827

We use the Version 1.0.5 in the lab which you can download from here:

<http://home.anadolu.edu.tr/~sgorgulu/micro2/gnu-m68k-win-1-0-5.exe>

The IDE68K tool also has C-support for C-based development (you can also use it), but it is recommended to learn how to use this gcc library. Because it gives the ability to handle programming a wide range of processors. In other words, if you learn how to use GCC, you will not be limited to only 68k, and you will be able to easily adapt yourself to program a wide range of processors.

First, install the GNU Embedded Development from the setup files above and restart your computer, then check out the **ReadMe.txt** and other document links in Start Menu program group (for Win XP).

After installation, if command-line executable tools do not run when prompted from a dos console windows, then you should add their path to the environment variables of your system.

Cited from ReadMe.txt:

Path - Modified to include the folders containing the main executable tools, typically:

```
C:\Program Files\MotoRobots\gcc-m68k\bin  
C:\Program Files\MotoRobots\gcc-m68k\vide
```

To modify system path (for WinXP), Right click on My Computer icon, and open Properties, in the Advanced tab, there is an Environment Variables button, push it. In the lower list, you should find the Path variable, and if still not added after installation, append the directory paths above to the end of the Path by ";"s between each. By adding those paths to the system path, you can call executables included in those folders from anywhere in system.

2. GNU Assembler

The gcc assembler syntax is a bit more complicated than the syntax used by as68k. Please refer to the documentation of GCC. We will become familiar with the basics by working with examples and studying the assembly code produced by the GNU C compiler.

```
.....
/* save this code as the file 'clab0.s' */

        .text
        .even
        .global main
main:
        clr.w sum          /* clear variable sum */
        move.w count,%d0  /* load count value */
loop:
        add.w %d0,sum     /* add d0 to sum */
        sub.w #1,%d0     /* decrement counter */
        bne loop        /* loop if counter not zero */
        move.w sum,%d7
        rts             /* return to caller */
sum:
        .space 2
count:
        .word 25

/* end of code for the file 'clab0.s' */
.....
```

This is a version of our lab program for the gcc assembler, m68k-elf-as. Note the following points:

- The syntax is closely related to C syntax.
- The program begins with a label called main, and this label is global so that it is visible to the linker **m68k-elf-ld**.
- main will be called by the startup routine which will be linked in. The startup routines in **crt0.s** sets up reset values for PC, SP, etc.
- Registers must always have a percent (%) in front.
- Note how variables and constants are set up.
- At completion, the program returns control to the startup routine using RTS.

3. Assembling and Linking

In order to assemble or compile programs for lab boards with the gcc tools, you will need to download and save the startup file **crt0.s** and the linker script **lab.ld** in your working directory (the same directory with your file). These two files together with the assembly program should all be in the same directory.

1. First, check with your tutor that the environment on your computer has been set up correctly.

2. Run the assembler in the command console window:

```
m68k-elf-as crt0.s clab0.s -o clab0.o -m68000 -ahls=clab0.lst
```

This will produce an object file **clab0.o** and a listing file **clab0.lst**

3. Next, run the linker:

```
m68k-elf-ld clab0.o -o clab0.s19 -L. -Tlab.ld
```

This will produce a Motorola s-record clab0.s19 which can be directly loaded onto the lab-board. This linking process links the startup routine and any other required routines from the standard C libraries together into one executable file.

Exercise 0:

- I. Connect to the lab-board and load the program executable clab0.s19. Run the code and check for correct operation.
- II. Find the memory location where the variable sum is stored, and check its contents.
- III. Examine the program listing in the memory viewer. Notice that the code which has been loaded into memory includes the startup routine and so your user program appears in the middle of this (translated). Closely examine the listing and memory contents. Explain. What addressing modes are used, and what offsets have been used in referring to count and sum.

4. First Example

Important Note: A dummy "void __main(){}" starter routine is necessary in c-codes to be compiled and linked correctly, and it should be inserted to each saved files hereafter.

Copy the C program clab1.c :

```
.....  
/* save this code ase file: clab1.c */  
int main()  
{  
    int i,j;  
    i = 1;  
    j = 2;  
    return i + j;  
}
```

.....
Examine the syntax, and determine what the program does. You can compile, assemble and link this program by typing the following commands in command console window:

```
m68k-elf-gcc -S clab1.c -o clab1.s -m68000  
m68k-elf-as crt0.s clab1.s -o clab1.o -m68000 -ahls=clab1.lst  
m68k-elf-ld clab1.o -o clab1.s19 -L. -Tlab.ld
```

Hints:

- o One can save those three elf-commands as an auto-batch file (ex: **allinone.bat**) using a text editor and instead of calling all those commands each time, call that file for once.
- o While you are using the bat-file; if you want to make it wait for your key-press after each command row, then insert the command "**pause**" between each row in your bat-file.

Exercise 1:

1. Load the program clab31.s19 into the lab board. Run the code and check for correct operation.
2. Examine the assembly code produced by the compiler in the file clab31.s (or in the listing file clab31.lst). What is the size of the int variables? How are they stored and manipulated? How is the stack frame used?

3. Edit the variable declaration line as follows:

```
register int i,j;
```

Recompile, assemble and link. Run the code and check for correct operation. How are the variables stored now? How is the stack frame used?

5. Subroutines

In this section, we will learn how GCC handles assembly language subroutines, parameter passing, and stack frames. We now look at some C programs with subroutines and see how the C compiler handles them.

```
.....  
/* save this code as the file: clab2.c */  
  
int adder(int x, int y)  
{  
    return x + y;  
}  
  
int main()  
{  
    int a, b, c;  
    a = 1; b = 2;  
    c = adder(a, b);  
}
```

Exercise 2:

1. Compile, assemble and link this program.
2. What does this program do?
3. Load the program clab32.s19 into the lab board. Run the code and check for correct operation.
4. Examine the assembly code produced by the compiler in the file clab32.s (or in the listing file clab32.lst). How are the variables stored in main and adder?
5. How are parameters passed to the subroutine? What are they? Are they passed by value or reference?

6. Pointers

We saw in previous lectures how parameters can be passed by value or reference. C uses pointers to pass parameters by reference. We look at several examples.

Copy the C program as clab3.c:

```
.....  
/* save this code as the file: clab3.c */  
  
#define number 12  
  
int adder(int x, int *py)  
{
```

```

        return x + *py;
    }

int main()
{
    int a, b, c;
    a = 1; b = number;
    c = adder(a, &b);
}

```

.....

Exercise 3:

1. What does this program do? Which variables are pointers?
2. Compile, assemble and link this program.
3. Load the program clab3.s19 into the board. Run the code and check for correct operation.
4. Examine the assembly code produced by the compiler in the file clab3.s (or in the listing file clab3.lst). How are the variables stored in main and adder?
5. How are parameters passed to the subroutine? What are they? Are they passed by value or reference?

7. Passing Addresses

The following program gives an example of a subroutine which attempts to swap numbers in variables.

Copy the C program as swap1.c:

```

/* save this code as the file: swap1.c */

void swap(int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}

int main()
{
    int x = 2, y = 3;
    swap(x, y);
    return 0;
}

```

.....

Exercise 4:

1. What does this program try to do? Are there any pointers?
2. Compile, assemble and link this program.
3. Load the program swap1.s19 into the BSVC simulator. Run the code and check for correct operation.

4. Examine the assembly code produced by the compiler in the file swap1.s (or in the listing file swap1.lst). How are the variables stored in main and swap?
5. How are parameters passed to the subroutine? What are they? Are they passed by value or reference?
6. Why doesn't this program work?

Copy the C program swap2.c:

```

.....
/* save this code as the file: swap2.c */

void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

int main()
{
    int x = 2, y = 3;
    swap(&x, &y);
    return 0;
}

```

.....

This program does what we want correctly.

Exercise 5:

1. What does this program do? Are there any pointers?
2. Compile, assemble and link this program.
3. Load the program swap2.s19 into the lab board. Run the code and check for correct operation.
4. Examine the assembly code produced by the compiler in the file swap2.s (or in the listing file swap2.lst). How are the variables stored in main and swap?
5. How are parameters passed to the subroutine? What are they? Are they passed by value or reference?
6. Why does this program work?

A WARM-UP LAB WORK FOR YOU

Estimating square root of a number without a floating-point library

Write a c-function to estimate the square root of a number given. The algorithm for finding a number's square root is by the formula on the right side. In the equation;

N is the number to be square rooted.

x_n is the current estimate, and

x_{n+1} is the next estimate.

$$x_{n+1} = \frac{\frac{N}{x_n} + x_n}{2}$$

Algorithm:

- Before the first iteration ($n = 0$), initial estimate (x_0) may be selected as half of N . For example, if N is 44, you may start with an initial estimate 22 (half of N). By the first iteration x_1 is calculated.
- In second iteration ($n = 1$), x_2 will be calculated using x_1 . And so on...

Therefore, the estimation (x_{n+1}) will be updated iteratively, and it will converge to actual value of the root. After iterating a few times (5 or more), one can get better resolutions. Multiply N and initial estimate each by 100 before iterations to get a good result whilst loss of remainders. Do not forget to reverse this magnification at the end.

Writing The Code

Your function's prototype may be one of the following. You can also use your own prototype.

```
int sqrt(int N, int noiterations); OR int sqrt(int &N, int noiterations);
```

Hint 1: By default, divide (/) and multiply (*) operators in your c code will not operate as expected. For these math. operators to work on m68k, you should define them. Write your own functions to handle multiplication and division operations. Remember that you can also merge assembly codes into your c code – (exercise 0).

Hint 2: Another option is to include some libraries which define math. operations. Therefore, let's dig into the GCC documentation to find such a library for m68000.

Sample code:

```
...
int sqrt(int N, int noiterations);
...
int main(void) /* not sure! */
{
    int mynum=44, nofi=6, result;
    ...
    result = sqrt(mynum, nofi);
    ...
    /* try to halt the program to see result through terminal.
    (without resetting the board) */
}
```

Test the code with at least three different numbers (60, 100, and 300) and iterations (4, 7, and 10).

Write a detailed report of this part including the usage of the code and results.