ARM Assembly Programming Using Raspberry Pi

1   Introduction

The Raspberry Pi is an inexpensive credit-card sized Linux computer. At its core is an ARMv6 CPU. The free download Raspbian package (from NOOBS http://www.raspberrypi.org/help/noobs-setup/) contains all the software necessary for ARM assembly language programming.

The downloaded package includes Raspbian operating system and several programming language supports. Among them is the GNU Compiler Collection (GCC) which supports programming in C, C++ and assembly languages.

In this document, we will use the commands **as** (assembler), **ld** (link loader), and **gdb** (GNU debugger) from GCC. These are command of command line interface that can be executed from the command prompt. If you are using the graphic user interface (GUI), the command prompt is available from LXTerminal which is part of the Raspbian software package.

We will assume the reader is comfortable using the command line interface of the Raspberry Pi. The Raspbian software package comes with two command line text editors: **nano** editor and **vi** that may be used to enter and edit the assembly source code. If you prefer to use a GUI text editor, Leafpad is available. For more text editor options , please visit http://www.raspberrypi.org/documentation/linux/usage/text-editors.md

Lastly, the screenshots in this document were captured remotely using Tera Term terminal emulator. The content should look identical to the console display of the Raspberry Pi.

Below are the versions of the assembler and linker used in this document.
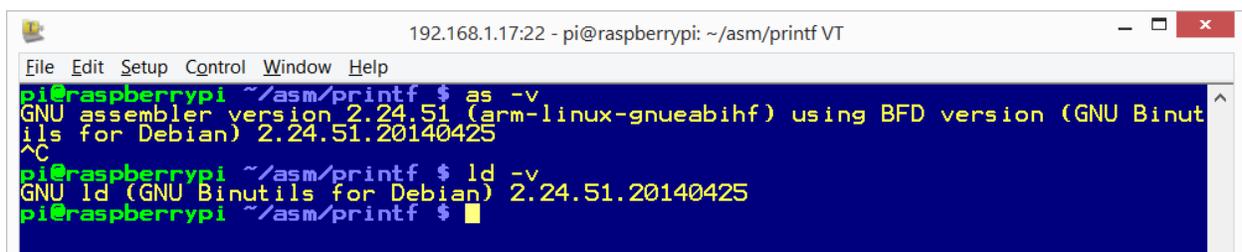
Figure 1: Display of version numbers for **as** and **ld** used in this document

2   The Differences

The example programs in the book were developed using Keil MDK-ARM and uVision IDE. These programs are intended to be stand-alone programs in an embedded ARM microcontroller. Programs developed using GCC tools in Raspberry Pi are applications running under the Raspbian OS. Here are some of the major differences:

1. The code and data of the program are all loaded in the RAM allocated by the OS using virtual memory addressing. The virtual memory address is the address that the program sees. From a programmer's perspective, it should make no difference unless you are attempting to read/write the hardware registers.
2. The program is running under Raspbian OS. The OS provides services such as console read/write or file I/O.
3. The syntax for the assembly source file is different. GCC was developed to support many different processors. With the recent version of GCC assembler (v2.24), ARM instructions are kept the same as Keil MDK-ARM, the other parts of the syntax are slightly different.
   a. Comments are preceded by '@' instead of ';'. The assembler also accepts C++ style comment enclosed with '/* */' and '//'.
   b. Labels must be followed by a ':'.
   c. The syntaxes of the directives are different but the mappings are straightforward. Below is a comparison table of the frequently used directives:

GCC Directive	Keil MDK-ARM Directive	Explanation
.text	TEXT	Signifies the beginning of code or constant
.data	DATA	Signifies the beginning of read/write data
.global *label*	GLOBAL or EXPORT	Makes the label visible to linker
.extern *label*	EXTERN	label is declared outside of this file
.byte *byte* [,*byte, byte, …*]	DCB	Declares byte (8-bit) data
.hword *hw* [,*hw, hw, …*]	DCW	Declares halfword (16-bit) data
.word *word* [,*word, word, …*]	DCD	Declares word (32-bit) data
.float *float* [,*float, float, …*]	DCFS	Declares single precision floating point (32-bit) data
.double *double* [,*double, double, …*]	DCFD	Declares double precision floating point (64-bit) data
.space *#bytes* [,*fill*]	SPACE or FILL	Declares memory (in bytes) with optional fill
.align *n*	ALIGN	Aligns address to <2 to the power of n> byte
.ascii "*ASCII string*"	DCB "*ASCII string*"	Declares an ASCII string
.asciz "*ASCII string*"	DCB "*ASCII string*", 0	Declares an ASCII string with null termination
.equ *symbol, value*	EQU	Sets the symbol with a constant value
.set *variable, value*	SETA	Sets the variable with a new value
.end	END	Signifies the end of the program

Table 1: Comparison of GCC directives and Keil MDK-ARM directives

3   Sample Program Conversion

Below is Program 2-1 from the book that was written in Keil MDK-ARM syntax.

<table>
<tr><td colspan="1" align="center">Program 2-1 Using Keil MDK-ARM syntax</td></tr>
</table>

```
; ARM Assembly Language Program To Add Some Data and Store the SUM in R3.


        AREA    PROG_2_1, CODE, READONLY
        ENTRY
        MOV    R1, #0x25      ; R1 = 0x25
        MOV    R2, #0x34      ; R2 = 0x34
        ADD    R3, R2, R1     ; R3 = R2 + R1
HERE    B      HERE           ; stay here forever
        END
```

The same program in GCC syntax for Raspberry Pi is below.

<table>
<tr><td colspan="1" align="center">Program 2-1p Using GCC as version 2.24 syntax</td></tr>
</table>

```
@P2_1.s ARM Assembly Language Program To Add Some Data and Store the SUM in R3.


          .global _start
_start:   MOV    R1, #0x25     @ R1 = 0x25
          MOV    R2, #0x34     @ R2 = 0x34
          ADD    R3, R2, R1    @ R3 = R2 + R1
HERE:     B      HERE          @ stay here forever
```

The changes are:

1. Comments either use C-style /* */ or are preceded with '@'.
2. Labels are followed by ':'.
3. GCC linker is expecting a label "_start" for the entry point of the program. This label also must be made global so that it is visible to the linker.

Technically the code of the program is marked by .text directive at the beginning and the end of the program should have the directive ".end". However, GCC does not enforce either one.

4   How to Assemble, Link and Run the Program

In this example, we enter the program above into a file name p2_1.s in the $HOME/asm directory, assemble, link, and execute the program.

First we make a directory with the name **asm**, change the current directory to **asm** and launch the editor vi for the file p2_1.s. We are showing the use of editor vi here but you may use any text editor you prefer.

Figure 2: make asm directory, change to asm directory and launch editor vi

After typing in the program in vi, the file is saved.

Figure 3: the sample program viewed in editor vi

The program is assembled using command "`as –o p2_1.o p2_1.s`". In this command, "`as`" is the name of the assembler, "`–o p2_1.o`" tells the assembler to create the output object file with the name p2_1.o, and lastly, "`p2_1.s`" is the assembly source file name (see below).

Figure 4: the assemble command

Like many Unix[1] programs, it produces no output to the console when the program ran without errors.

---

[1] Raspbian is ported from Debian which derived from Linux and Linux is derived from Unix. All of them are very similar. In this document we will use Unix as a generic term for these operating system.

Linker takes one or more object files and creates an executable file. To run the linker, use command "`ld –o p2_1 p2_1.o`". In this command, "`ld`" is the name of the linker program, "`–o p2_1`" tells the linker to produce the output executable file with the name p2_1, and lastly, "`p2_1.o`" is the input object file name.

Figure 5: the linker command

Again, the linker produces no output to the console when there were no errors.

To execute the program, type the command "`./p2_1`" at the prompt. It tells the shell to execute the program named **p2_1** at the current directory.

Figure 6: the command to execute the program p2_1

Recall the last instruction in the program is an infinite loop. After executing the first three instructions, the program is stuck at the infinite loop that consumes 100% of the CPU time. An infinite loop is typical for a program in a simple embedded system without an operating system. For now, type Ctrl-C to terminate the program and get the prompt back.

Figure 7: Ctrl-C to terminate the program

5   Program Termination

As an application running under a multitasking operating system, the program should be terminated when done. Otherwise, the program running a dummy infinite loop will consume the CPU time and slow down all the other programs.

To terminate a program, replace the dummy infinite loop at the end of the program "**HERE: B HERE**" by:

```
MOV    R7, #1
SVC    0
```

The number 1 placed in Register 7 tells the operating system to terminate this program. The instruction "`svc 0`" is the **system call**, that transfers the program execution to the operating system. If you place a different number in R7, the operating system will perform a difference service. We will visit a system call to write to the console later.

After replacing the dummy infinite loop with the system call to end the program, run the assembler and linker again. This time after you execute the program, the program will terminate by itself and the prompt reappears immediately without user intervention.

6   Using GDB

A computer without output is not very interesting like the previous example program. We will see how to generate output from a program in the next section, but for most of the programs in this book, they demonstrate the manipulations of data between CPU registers and memory without any output. GDB (GNU Debugger) is a great tool to use to study the assembly programs. You can use GDB to step through the program and examine the contents of the registers and memory.

In the following example, we will demonstrate how to control the execution of the program using GDB and examine the register and memory content.

6.1   Preparation to Use GDB

When a program is assembled, the executable machine code is generated. To ease the task of debugging, you add a flag "`-g`" to the assembler command line then the symbols and line numbers of the source code will be preserved in the output executable file and the debugger will be able to link the machine code to the source code line by line. To do this, assemble the program with the command ('$' is the prompt):

```
$ as -g -o p2_1.o p2_1.s
```

The linker command stays the same:

```
$ ld -o p2_1 p2_1.o
```

To launch the GNU Debugger, type the command gdb followed by the executable file name at the prompt:

```
$ gdb p2_1
```

After displaying the license and warranty statements, the prompt is shown as (gdb). See Figure 8 below.

Figure 8: assemble with debug option and launch of gdb

6.2 Exit GDB

GDB has a wealth of commands. We will only touch a few here. At the gdb prompt, you only need to type the minimal number of characters to be distinctive for the command. We will show the full command but underscore the minimal required characters.

To exit GDB, the command is "quit".

6.3 List source code

To list the source code, the command is "list". The list command displays 10 lines of the source code with the line number in front of each line. To see the next 10 lines, just hit the Enter key.

Figure 9: use gdb command **list** to display source code with line number

6.4 Set breakpoint

To be able to examine the registers or memory, we need to stop the program in its execution. Setting a breakpoint will stop the program execution before the breakpoint. The command to set breakpoint is

"**b**reak" followed by line number. The following command sets a breakpoint at line 6 of the program. When we run the program, the program execution will stop right before line 6 is executed.

```
$ b 6
```

GDB will provide a confirmation of the breakpoint.

```
8        MOV     R2, #0x34        @ R2 = 0x34
7        ADD     R3, R2, R1       @ R3 = R2 + R1
8  HERE:  B       HERE            @ stay here forever
(gdb) b 6
Breakpoint 1 at 0x8058: file p2_1.s, line 6.
(gdb)
```

Figure 10: set a breakpoint at line 6 of the source code

6.5 Run the program

To start the program, use command "**r**un". Program execution will start from the beginning until it hits the breakpoint. The line just following where the breakpoint was set will be displayed. Remember, this instruction has not been executed yet.

```
4  _start:
5        MOV     R1, #0x25        @ R1 = 0x25
6        MOV     R2, #0x34        @ R2 = 0x34
7        ADD     R3, R2, R1       @ R3 = R2 + R1
8  HERE:  B       HERE            @ stay here forever
(gdb) b 6
Breakpoint 1 at 0x8058: file p2_1.s, line 6.
(gdb) r
Starting program: /home/pi/asm/p2_1

Breakpoint 1, _start () at p2_1.s:6
6        MOV     R2, #0x34        @ R2 = 0x34
(gdb)
```

Figure 11: run the program and it stops at the breakpoint

6.6 Examine the CPU registers

With the program stopped before line 6, the last instruction on line 5 moved a literal value 0x25 into Register 1. We can verify that by using command "**i**nfo **r**egisters". The display consists of three columns: the register name, the contents in hexadecimal, and the contents in decimal. The registers holding addresses such as SP or PC will not display decimal values.

Figure 12: display register contents using info register command

6.7 Disassemble machine code

GDB has the ability of disassembling the machine code back to assembly instructions. The command is "**disas**semble". Because the assembler does more than translating the source code to machine instructions, the disassembled result may differ from the original source code. For example, a pseudo-instruction entered as "**ldr R5, =0x1234**" in the source code may have the disassembled output as "**ldr r5, [PC, #32]**".

Figure 13: disassemble the machine code

In the disassembled display, the breakpoint instruction is marked by an arrow at the left margin.

The disassemble command also takes a pair of starting address and ending address separated by a comma such as:

```
$ disas 0x8054, 0x806c
```

Note in the example in Figure 14 that although the last instruction of the program is at address 0x00008060, the disassembled output continued until the specified address was met.

```
End of assembler dump.
(gdb) disas 0x8054,0x806c
Dump of assembler code from 0x8054 to 0x806c:
   0x00008054 <_start+0>:        mov     r1, #37 ; 0x25
=> 0x00008058 <_start+4>:        mov     r2, #52 ; 0x34
   0x0000805c <_start+8>:        add     r3, r2, r1
   0x00008060 <HERE+0>: b        0x8060 <HERE>
   0x00008064:   andeq   r1, r0, r1, asr #6
   0x00008068:   cmnvs   r5, r0, lsl #2
End of assembler dump.
(gdb)
```

Figure 14: disassemble the machine code between address 0x8054 and 0x806C

6.8   Step through the instructions

When the program execution is halted by the breakpoint, we may continue by stepping one instruction at a time by using command "**stepi**". The step instruction command may also take a numeric argument to step more than one instruction at a time. For example, "**stepi 5**" will execute the five instructions or until another breakpoint is hit.

In the example below in Figure 15, we stepped two instructions and examined the register contents.

```
(gdb) stepi
7               ADD     R3, R2, R1        @ R3 = R2 + R1
(gdb) stepi
HERE () at p2_1.s:8
8       HERE:   B       HERE              @ stay here forever
(gdb) i r
r0              0x0             0
r1              0x25            37
r2              0x34            52
r3              0x59            89
r4              0x0             0
r5              0x0             0
r6              0x0             0
```

Figure 15: step instruction twice and examine the register content

6.9   Continue program execution

When the program execution is stopped, you may continue from where execution was halted by command "**c**ontinue".

In this example as seem in Figure 16, if we continue program execution and there are no more breakpoints left, the program will run without stopping and we have no gdb prompt to issue a command. To stop program execution, hit Ctrl-C.

```
(gdb) cont
Continuing.
^C
Program received signal SIGINT, Interrupt.
HERE () at p2_1.s:8
8       HERE:   B       HERE              @ stay here forever
(gdb)
```

Figure 16: continue the program execution and terminated by Ctrl-C

6.10 Examine the memory

The command to examine the memory is "**x**" followed by options and the starting address. This command has options of length, format, and size (see Table 2: options for examine memory command). With the options, the command looks like "**x**/nfs *address*".

Options	Possible values
Number of items	any number
Format	**o**ctal, he**x**, **d**ecimal, **u**nsigned decimal, bi**t**, **f**loat, **a**ddress, **i**nstruction, **c**har, and **s**tring
Size	**b**yte, **h**alfword, **w**ord, **g**iant (8-byte)

Table 2: options for examine memory command

For the example in Figure 1Figure 17 to display eight words in hexadecimal starting at location 0x8054, the command is "`x/8xw 0x8054`".

```
cpsi         0x10      18
(gdb) x/8xw 0x8054
0x8054 <_start>:      0xe3a01025      0xe3a02034      0xe0823001      0xeaffff
fe
0x8064: 0x00001341    0x61656100      0x01006962      0x00000009
(gdb)
```

Figure 17: example of examine memory command

7   Floating Point

The Raspberry Pi has hardware floating point support (VFP). You may write 32-bit or 64-bit floating point instructions in the assembly program.

To assemble a program with floating point instructions, you need to let the assembler know that you are using the VFP instructions by adding the command line option "-mfpu=vfp". A sample command will be:

`$ as –mfpu=vfp –g –o p.o p.s`

The linker command remains the same.

In GDB, you may examine the VFP registers using command "**i**nfo **fl**oat". The registers are displayed in 64-bit mode and 32-bit mode with floating point output and hexadecimal output. The content of the VFP status register fpscr is also displayed.

8   Program Output

8.1   Program Return Value

Although most of the programs in the book manipulate data internal to the processor and memory, it is still convenient to generate output to be observed outside of the debugger. Recall in section 5 we discussed terminating a program using system call #1. Following the Unix convention, when a program terminates, it should return an exit code. In an assembly program the exit code should be left in Register 0 before the exit system call is made. After the program exit, the user may retrieve the exit code by reading the shell variable "$?". This is a simple method for a single integer output of a program.

Using the last sample program, we will send the result of the addition to output. We do so by placing the result in R0 before making the exit system call. The end of the program will look like this:

```
    MOV     R0, R3
```

```
        MOV      R7, #1
        SVC      0
```

After the program exit, at the prompt type command "echo $?" and the result 89 (decimal) will be displayed.

Figure 18: an example of producing a program output number

9 Assembly Programming with GUI

So far, we have been using command line interface for programming and debugging. The rest of this document will describe the use of a graphic user interface, Code::Blocks IDE, for assembly programming.

Code::Blocks (http://www.codeblocks.org/) is an open source C, C++ and Fortran IDE licensed under the term of GNU General Public License version 3. Although Code::Blocks is not intended for assembly language programming, with GNU GCC compiler and GNU GDB debugger plug-ins, it can handle assembly language programming.

Code::Blocks IDE does not support an assembly language only project so we need to start a C project then replace the "main.c" with an assembly language source file. (I will support a project with both C language source files and assembly language source files but that is beyond the scope of this document.) As a C language project, the C start-up code will be automatically linked into the executable code but this will be mostly transparent to the assembly programmer (or C programmer for that matter). We will describe the differences it causes later.

9.1 Install Code::Blocks IDE

To install Code::Blocks IDE, use the following command at the command prompt and all the required software will be installed.

```
$ sudo apt-get install codeblocks
```

You do need Internet connection to download and install the software package and it will take several minutes to complete.

9.2 Start Code::Blocks IDE

To start the Code::Blocks IDE, click the start button (at the lower left corner of the Desktop), select Programming then Code::Blocks IDE. (To bring up the Desktop from command line interface, type command "**startx**".)
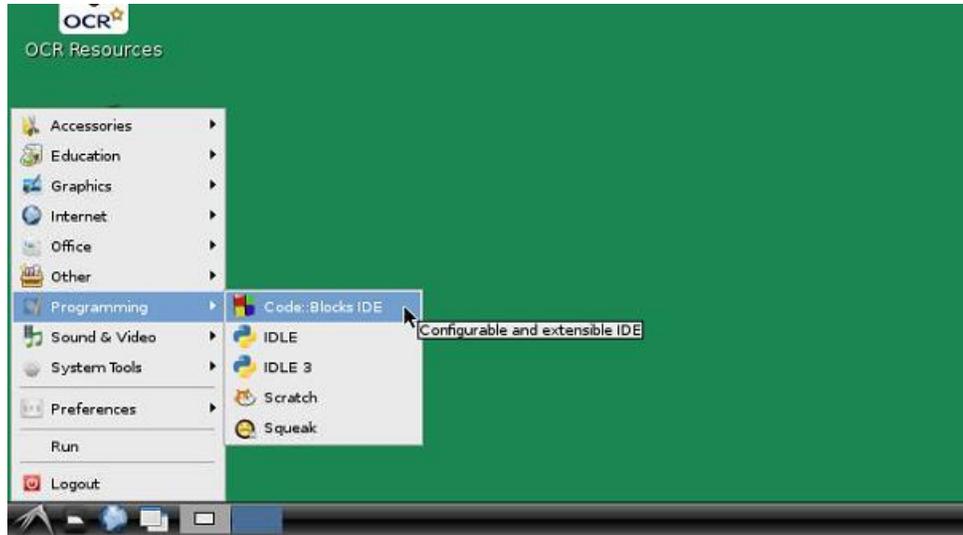
Figure 19: start Code::Blocks IDE

The first time you launch Code::Blocks IDE, it will ask you to confirm the compiler plug-ins. Unless you have installed other compiler tools, it should detect only the GNU GCC compiler.

9.3 Create a Code::Blocks Project

9.4 Using Project Wizard to Create a Project

After the IDE is launched, it displays the "Start here" window. Click on "Create a new project" link and a window with the title "New from template" pops up.

Keep the default wizard type as "Project" and click select "Console application" wizard then click the "Go" button (see Figure 20).
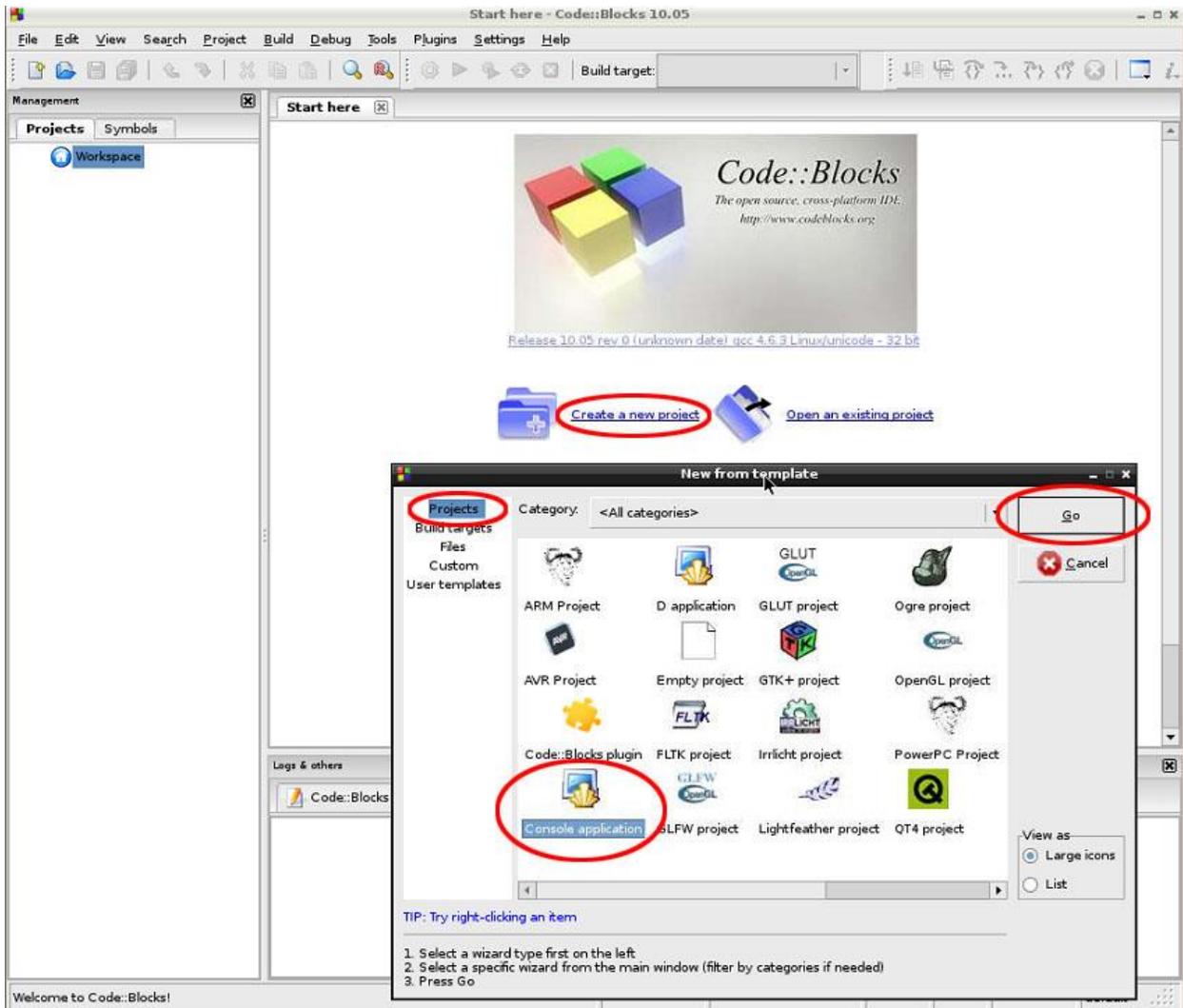
Figure 20: create a new project using "Console application" wizard

The window will be replaced by "Console application" wizard. First it will ask for the selection of programming language. Select "C" then click "Next>" button (See Figure 21).

Figure 21: select C language for the project

The next window asks for the project title and the folder where the project will be created. The wizard creates a folder with the project name under the folder you specified and put all the project files and folders in it. For example, we selected a folder "/home/pi/asm" and a project title "p2_1" (See Figure 22). A folder "/home/pi/asm/p2_1" will be created and all the files and folders of this project will be placed in it.

Figure 22: specify project title and project folder location

Click "Next>" button and the next window is used to select compiler and configuration. Take the default selections as in Figure 23 then click "Finish" button.
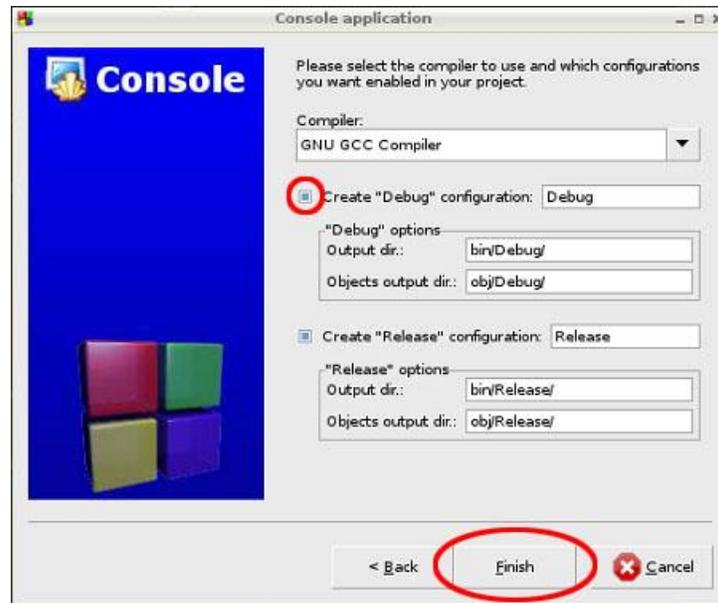
Figure 23: select compiler and configurations

9.5 Replace the C File with an Assembly File

When we created a project for C language, the project wizard put in a template C source file called "main.c". We will remove it and replace it with an assembly source file.

Expand the project tree in the left panel and right click on "main.c" then select "Remove file from project" (See Figure 24).
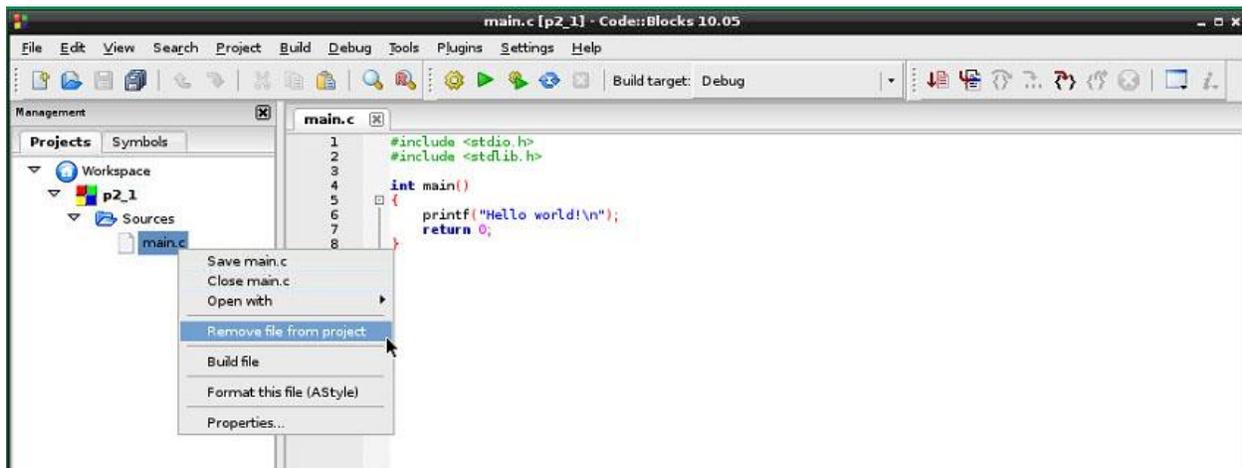
Figure 24: remove main.c from the project

Now, add an empty file to the project from menu File->New->Empty file (See Figure 25).

Figure 25: add an empty file to the project

Click "Yes" button to confirm adding this file to the project (See Figure 26).
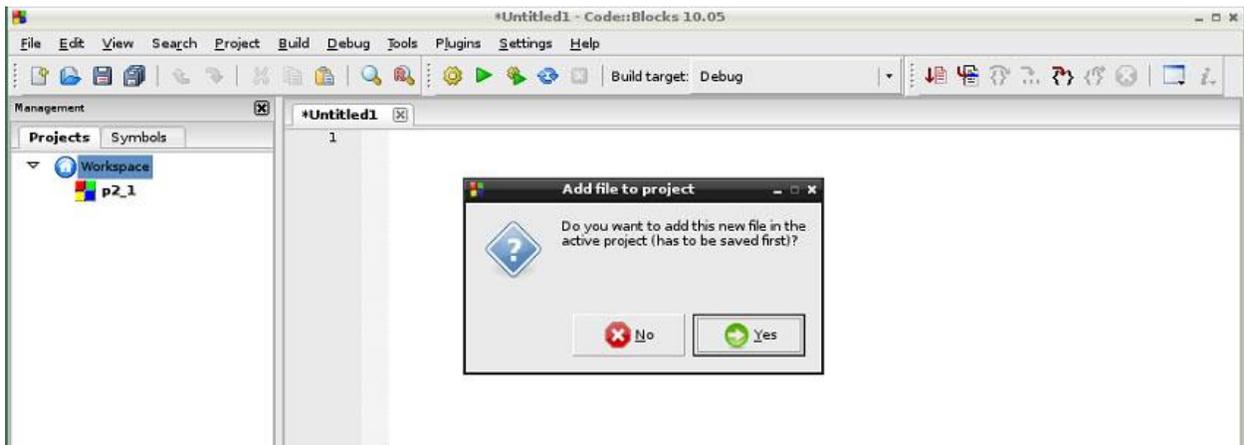
Figure 26: click "Yes" to add the file to the project

In order to add a file to the project, the file needs to be saved. The "Save file" window will pop up. Enter a file name (for example, p2_1.s) and click "Save" button (See). The file needs an extension of ".s" to be recognized as an assembly language source file.
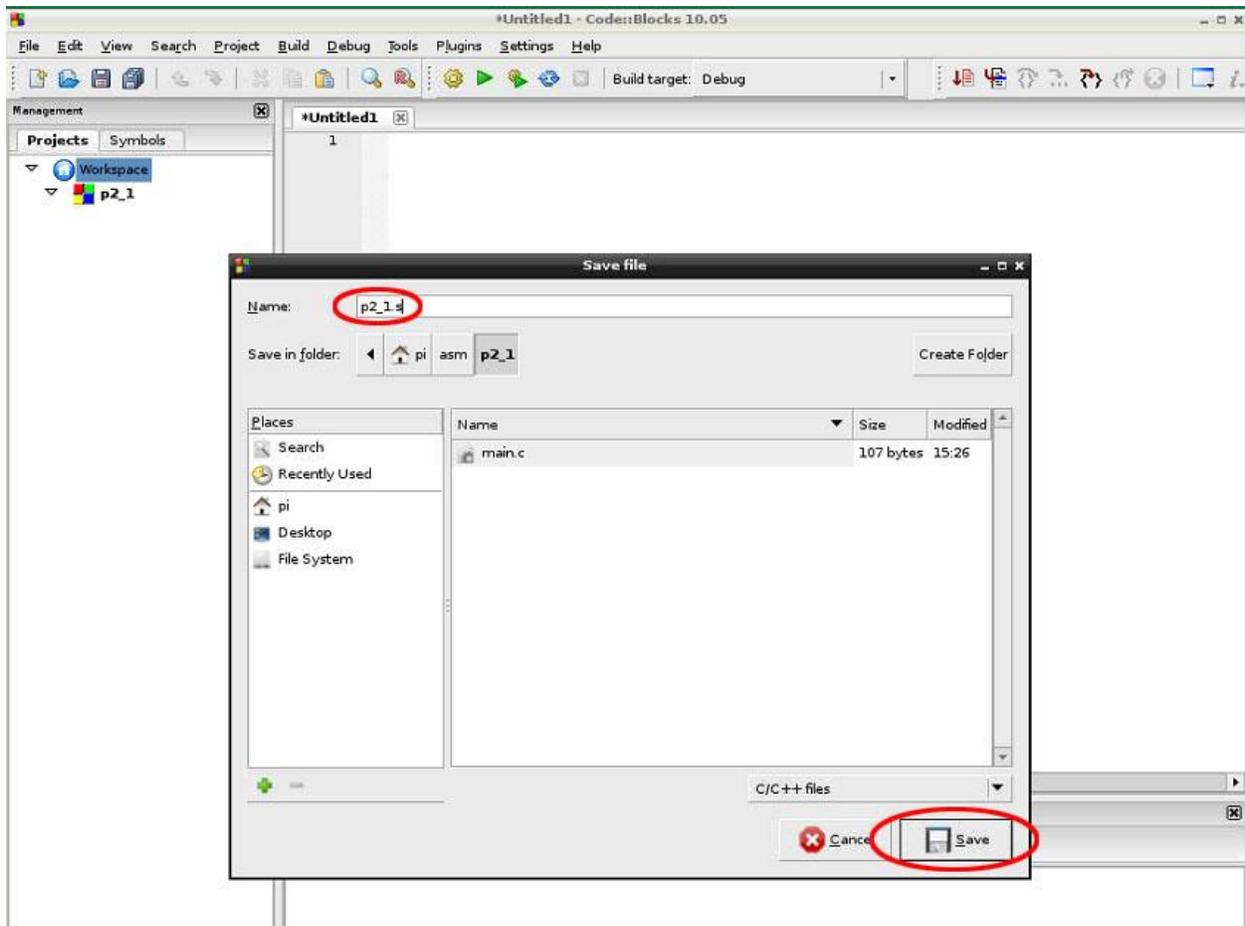
Figure 27: save the file with a file name

The next window pops up will ask you to select the targets for this file (See Figure 28). Make sure "Debug" is selected before clicking on the "Ok" button.

Figure 28: select target for the file

Once the empty file is added to the project, we may type in the code. We will reuse the program example from the early command line interface with a small change.

Program 2-1g Using GCC as version 2.24 syntax

```
@P2_1.s ARM Assembly Language Program To Add Some Data and Store the SUM in R3.


            .global main
main:       MOV    R1, #0x25    @ R1 = 0x25
            MOV    R2, #0x34    @ R2 = 0x34
            ADD    R3, R2, R1   @ R3 = R2 + R1
HERE:       B      HERE         @ stay here forever
```

Recall the GCC linker is expecting a label "_start" as the entry point of the program. That's why we used it in the previous program. The difference here is that we are borrowing a C language project for our assembly program. When GCC is linking a C program, it includes the C startup code at the beginning of the program. The C startup code has the label "_start" as the entry point and the program execution starts there. At the end of the C startup code, the program branches to a label "main" so we need to use "main" as the label of the entry point of our program.

9.6 Assemble the program

To run the assembler and linker, click the "Build" button (Figure 29).
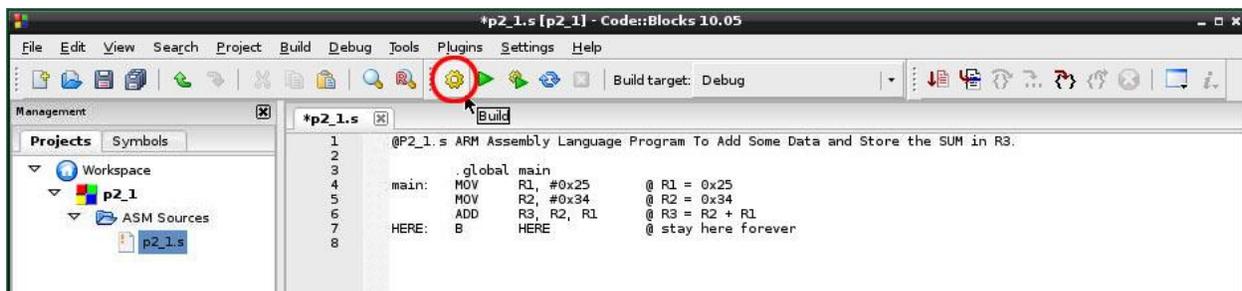
Figure 29: click "Build" button to assemble and link the program

The "Build log" window will show at the lower part of the window. Make sure there are no errors nor warnings (See Figure 30).
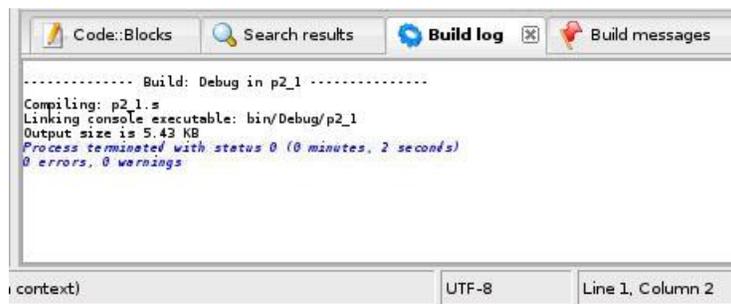
Figure 30: Build log display

9.7 Breakpoint and Launch of the Debugger

Before we launch the debugger, it is a good idea to set a breakpoint. Otherwise, the debugger will run all the way to the end of the program. To set a breakpoint, click at the margin just to the right of the line

number in the source editor window and a red stop sign will show. To remove a breakpoint, click on the red stop sign and the breakpoint will be removed.

With the breakpoint set, click "Debug / Continue" button to launch the debugger (See Figure 31).
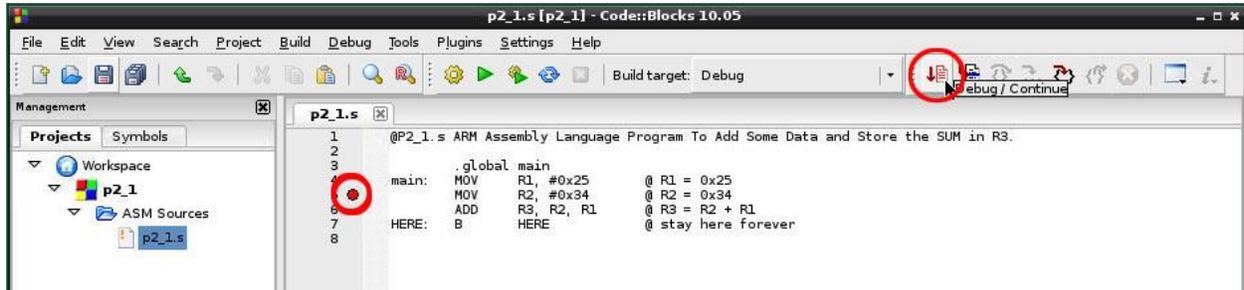
Figure 31: breakpoint and "Debug" button

The debugger changes the perspective of IDE then run the program until a breakpoint is hit. In our program with the breakpoint on line 5, the program executes until right **before** the instruction on line 5 is executed.
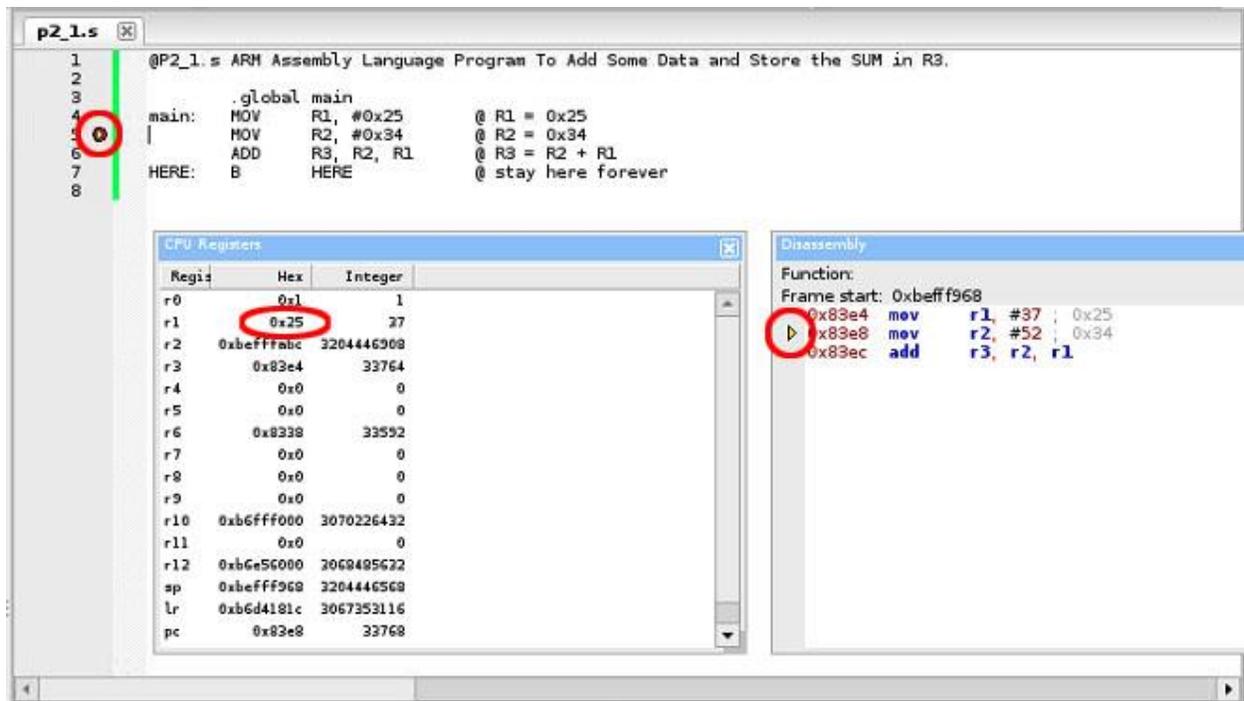
Figure 32: the debugger stops before the breakpoint line is executed

From Figure 32 above, you may see that there is a yellow triangle superimposed on the stop sign on line 5 denotes that the program execution is halted and the program counter is pointing to the instruction on line 5. The same is shown in the Disassembly window. The CPU Register window shows that r1 has the content of 0x25 as the result of the instruction on line 4.

If Register window or Disassembly window is not visible, you may enable them from menu "Debug->Debugging windows".

9.8 Stepping the Instruction

When the program execution is halted, the "Next instruction" button may be used to step the program one instruction at a time ().
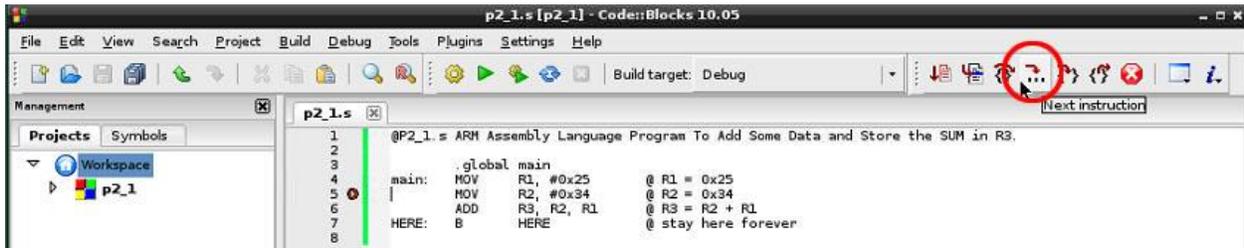
Figure 33: the "Next instruction" button

When the "Next instruction" button is pressed, the yellow arrow moves to the next instruction (line 6 in Figure 34). Also as the result of instruction on line 5, register r2 has the content changed to 0x34..
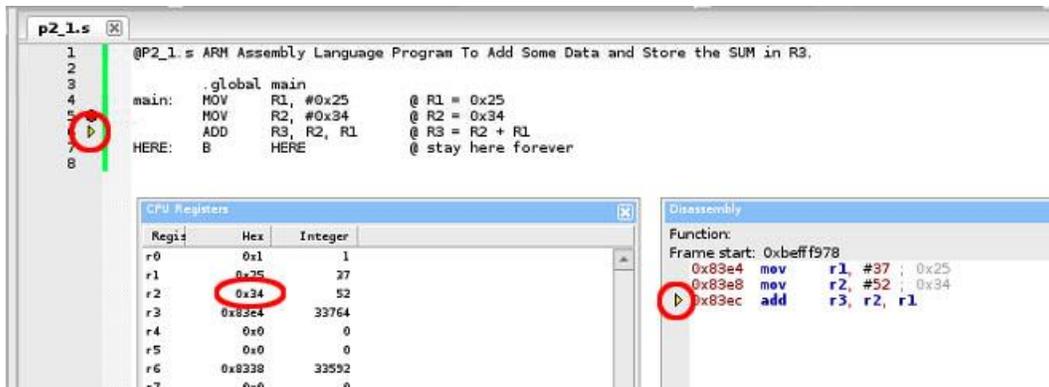
Figure 34: the result of "Next instruction"

9.9 Examine Memory

The memory window can be enabled from menu "Debug->Debugging windows->Examine memory" (Figure 35).
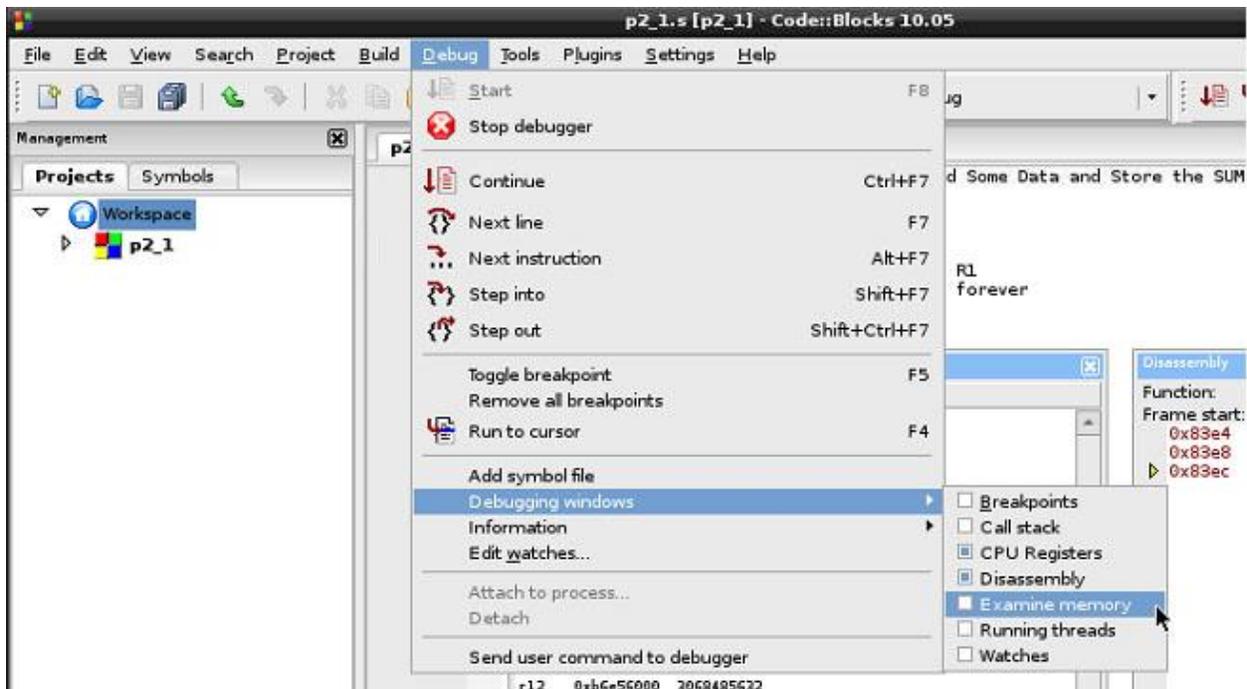
Figure 35: enable Examine memory window

9.10 Floating Point Registers

The floating point registers can be examined from menu "Debug->Information->FPU status" (Figure 36).
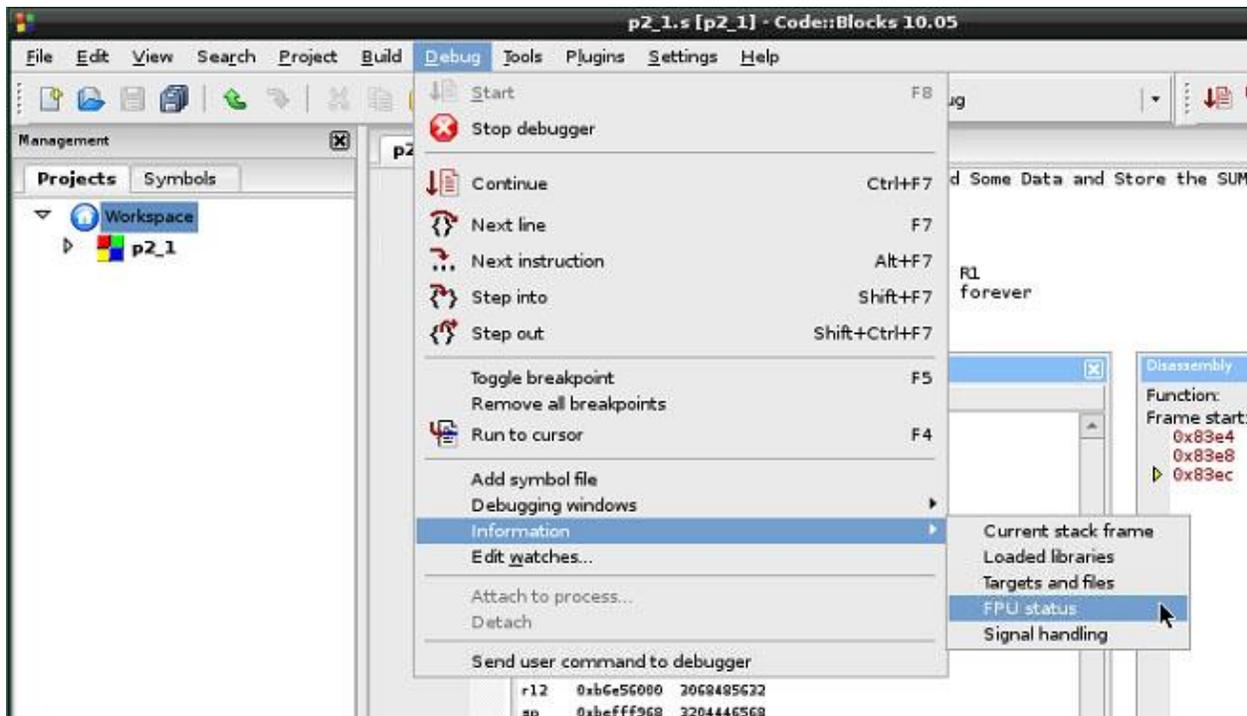
Figure 36: display FPU status

Bear in mind, the information displays are modal (when one of them is opened, you may not interact with the debugger until it is closed).

9.11 Stop the Debugger

To stop the debugger and return to Edit/Build perspective of IDE, click "Stop debugger" button (Figure 37).
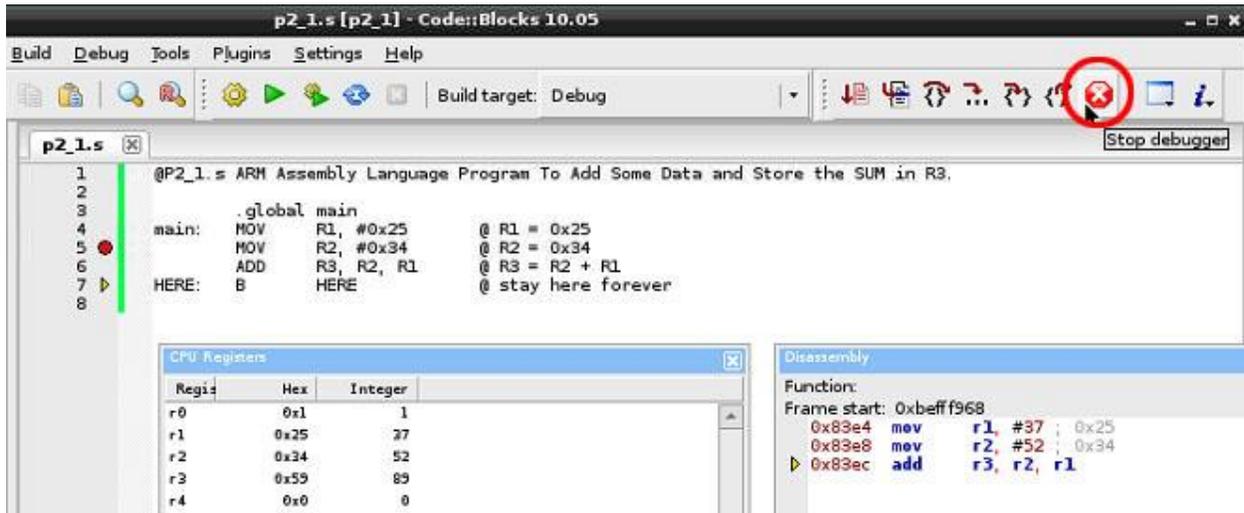
Figure 37: Stop debugger button

10 Conclusion

This document described how to use GCC tools to write assembly programs in a Raspberry Pi. It also discussed the use of GDB debugger. Lastly, the use of Code::Blocks IDE as the GUI is introduced.