



Getting Started with Arm Assembly Language

Version 1.0

Non-Confidential

Copyright © 2022 Arm Limited (or its affiliates).
All rights reserved.

Issue 01

107829_0100_01_en



Getting Started with Arm Assembly Language

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	21 December 2022	Non-Confidential	Initial release

Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly

or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on <https://support.developer.arm.com>

To provide feedback on the document, fill the following survey: <https://developer.arm.com/documentation-feedback-survey>.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview.....	7
2. Before you begin.....	8
3. What is assembly language?.....	9
3.1 Machine code.....	9
3.2 Other programming languages.....	9
3.3 How assembly code works.....	10
3.4 Why use assembly?.....	11
4. Calling an assembly function from C code.....	12
5. Assembly language basics.....	15
5.1 Registers.....	15
5.2 Loading data into registers.....	15
5.3 Program flow.....	17
5.4 Arithmetic operations.....	17
5.5 A64 instructions.....	18
6. Investigate registers with Arm Development Studio.....	19
7. Test instructions, branching, and loops.....	23
7.1 Test instructions.....	23
7.2 Test flags (NZCV).....	23
7.3 Branching and loops.....	24
8. Example: decrement until equal.....	25
9. Investigate NZCV flags with Arm Development Studio.....	27
10. Example: equation calculation.....	29
11. Example: factorial calculation.....	32
12. Related information.....	35

13. Next steps.....	36
----------------------------	-----------

1. Overview

This guide introduces the basic concepts of Arm assembly language using the A64 instruction set, shows you how to create and run assembly code using Arm Development Studio, and provides examples of assembly code for you to experiment with.

Assembly language is a low-level programming language, just one step above the processor's native language, machine code.

Writing an entire program in assembly language, even a relatively simple one, is complicated. That is why most people use high-level languages like C or C++ to write their programs, and then use a compiler to turn that high-level program into machine code.

For this guide, we want to write some assembly language to learn how it works. But it would still be too much work to write an entire program in assembly by hand.

So, the solution is to combine the best of both worlds. We will write the main program in C, and that program will call out to a function written in assembly code. We can then write our own assembly code in the function, without needing to worry about all the other code needed to make the program run.

We will then use this method to look at different examples of simple programs written in Arm assembly language. We will show how these examples work, and investigate how different features of the Arm architecture let us write assembly programs that can perform any task you can imagine.

2. Before you begin

Before you follow the examples in this guide, you need to download Arm Development Studio and get a simple C program running on a Fixed Virtual Platform (FVP) model.

Arm Development Studio is a professional software development solution for bare-metal embedded systems and Linux-based systems. Among other features, Arm Development Studio includes Arm Debugger, Arm Compiler, and built-in FVPs. Arm recommends Arm Development Studio for the procedures and examples in this guide.

Do the following:

1. Download the latest version of [Arm Development Studio](#).

See the [Arm Development Studio Getting Started Guide](#) for more information on installing Arm Development Studio.

2. Run the Hello World example provided with Arm Development Studio.

Follow all the steps in the [Hello World tutorial](#) in the Arm Development Studio Getting Started Guide.

The [Hello World tutorial](#) provides a simple C code example program, and shows you how to compile and run it on an FVP model of an Arm processor.



Although you can run this example in C or C++, the following examples in this guide will only work with C.

Once you have the basic Hello World programming running, you can continue with the rest of this guide.

3. What is assembly language?

This section of the guide provides information about why you would want to program in assembly language when other programming languages, like C, are much easier to write and understand.

3.1 Machine code

Before we talk about assembly language, let's consider another language: machine code.

Machine code is the fundamental language of computing. It is the only native language a processor understands. All other programming languages must be converted into machine code before the processor can run them.

A processor runs machine code instructions by looking at the raw 1s and 0s in the code, and uses those values to switch the on-or-off status of transistors in the processor itself. These switches control the behavior of the processor, and make the processor perform the operation intended by the machine code instruction.

Writing machine code by hand is very difficult for humans. Writing code with individual 1s and 0s is similar to writing a novel using the dots and dashes of [Morse code](#). So, we have developed programming languages that abstract machine code at various levels in order to make programming easier.

3.2 Other programming languages

We can think about different programming languages in terms of their abstraction level: how far removed they are from machine code, the native language of the processor:

- High-level languages

Languages like C offer a high level of abstraction: you can program relatively complex algorithms using a small number of C commands. A special program called a compiler then turns C programs into machine code. A C program containing just a handful of commands might easily produce machine code containing hundreds of instructions. Languages like C are called high-level languages.

- Assembly language

Assembly language is essentially a representation of machine code in human-readable words. It is just one small step of abstraction above machine code. Each assembly code instruction usually corresponds to a single machine code instruction. It provides human-readable instructions which map directly to the 1s and 0s of machine code.

For example, rather than using the machine code instruction “1001000100”, you can use the `add` instruction. They both mean the same thing, but the assembly instruction is much easier to read.

A program called an assembler converts the assembly code into machine code. Because assembly instructions directly correspond to machine code instructions, an assembler is a much simpler program than a compiler.

3.3 How assembly code works

Assembly language is a layer just above machine code which makes it easier for humans to read and write.

For example, consider the following line of assembly code:

```
add x1, x0, #1
```

This assembly instruction consists of the operation code (or opcode) `add` plus three operands, `x1`, `x0`, and `#1`.

For most assembly language instructions, the opcode dictates what the instruction does (adds two numbers together), and the operands specify which values are used. In this example, the instruction adds 1 to the value in the `x0` register, and then puts the result in the `x1` register.

Different instructions have different numbers and types of operand, and use those operands in different ways depending on what the instruction actually does. For a full description of available instructions and their operands, see [A64 base instructions](#).

The assembler converts the assembly language instruction to the following machine code:

```
1001000100000000000000010000000001
```

This can be broken down into its components:

```
1001000100      <- Operation code for the add instruction
   000000000001  <- immediate value to add, #1
             00000 <- source register, x0
             00001 <- destination register, x1
```

You can see the machine code equivalent for all assembly instructions in the A64 Instruction Set Architecture documentation. For example, [here is the section](#) that describes with this `add` instruction.

Hopefully you can already see that programming in assembly language is going to be much easier to understand than programming directly in machine code!

3.4 Why use assembly?

Assembly language instructions are very basic in comparison to the instructions in higher-level programming languages like C. But we can build any complex operation you can think of from these basic assembly language instructions. We just need to use combinations of multiple instructions.

Because assembly language instructions correspond directly with machine code instructions, it is often impractical to write a whole program using assembly code. It's usually much easier to write your program in a high-level language like C and let the compiler do the tedious work of converting that C program into the hundreds, thousands, or millions of resulting machine code instructions.

Scenarios where writing assembly by hand might be useful include the following:

- You might think that you can write more efficient assembly code than the compiler, and want to hand-optimize critical functions in your code to improve performance. With modern compilers, these situations are rare though.
- A much more common scenario is people wanting to write programs to help them learn. Writing assembly code by hand is a great way to learn about the Arm instruction set, and to experiment with it. If this is your motivation, we hope that this guide provides a useful place to start.
- Another situation where you might want to use assembly code is configuring system registers that can only be accessed using assembly.
- Being able to read assembly code can help when debugging complex code.

There are, however, some disadvantages to writing in assembly language:

- Not portable across different computer architectures.
- Source code is more difficult to read and maintain.
- Requires knowledge of the computer system and resources.

4. Calling an assembly function from C code

Writing an entire program in assembly language is hard work. That is why most people use high-level languages like C or C++ to write their programs, and then use a compiler to turn that high-level program into machine code.

For this guide, we want to write some assembly language to learn how it works. But it would still be too much work to write an entire program in assembly by hand.

So, the solution is to combine the best of both worlds. We will write the main program in C, and that program will call out to a function written in assembly code. We can then write our own assembly code in the function, without worrying about all the other code needed to make the program run.

If you have not already done so, start the Arm Development Studio IDE and create the HelloWorld C project by following all the steps in the [Hello World tutorial](#) from the Arm Development Studio Getting Started Guide.

To modify the HelloWorld project to call an assembly code function, do the following:

1. Create a new assembly code file.
 - a. In Project Explorer, right-click the `src` folder in the HelloWorld project and select New > File.
 - b. In the Create New File dialog, use the File Name field to name this file `my_assembly.s` and click Finish. The empty script opens in the Editor window.
 - c. In the `my_assembly.s` Editor window, enter the following code:

```
.global    my_function
.type     my_function, "function"
.p2align  4
my_function:
    add     x0, x0, x1
    ret
```

The `.global` directive marks the symbol `my_function` as a global symbol. This allows `my_function` to be referenced by our main C code.

The `.type` directive sets the type of the symbol `my_function` to function. This allows our main C code to call `my_function` as a function.

The `.palign` directive ensures that our code is properly aligned to an 8-byte boundary.

We will take a look at how the rest of this assembly code works in [Assembly language basics](#).

2. Open the `HelloWorld.c` file in Arm Development Studio.

In Project Explorer, double-click `HelloWorld.c` in the `HelloWorld/src` folder. The `HelloWorld.c` file opens in the Editor window.

3. In the Editor window for `HelloWorld.c`, enter the following code:

```
#include <stdio.h>

extern int my_function(int a, int b);

int main()
{
    int a = 4;
    int b = 5;
    printf("Calling assembly function my_function with x0=%d and x1=%d results\n", a, b, my_function(a, b));
    return (0);
}
```

4. Build the HelloWorld project.

In the Project Explorer view, right-click the HelloWorld project and select Build Project.

The build uses the Arm Compiler for Embedded 6 compiler to do the following:

- Compile the C code to machine code
- Assemble the A64 code to machine code, using the integrated assembler
- Link the separate machine code portions together to create an executable we can run

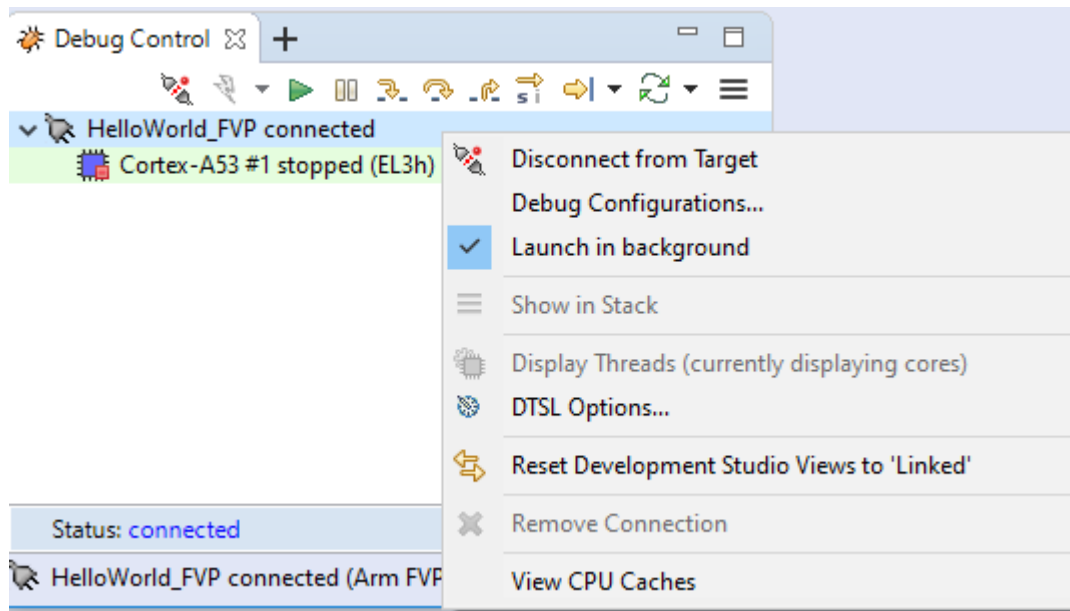
When the build has completed, a message similar to the following is shown in the Console window:

```
11:32:54 Build Finished. 0 errors, 0 warnings. (took 927ms)
```

5. Run your program on the FVP model.

In the Debug Control view, right-click HelloWorld_FVP and select Connect to Target.

If the FVP is already running, disconnect it first by selecting Disconnect from Target:

Figure 4-1: Disconnect HelloWorld_FVP

The application loads on the target, and stops at the `main()` function, ready to run.

6. In the Debug Control view, click the green arrow icon to continue running the application.

Arm Development Studio runs the applications, then when it finishes switches to the Disassembly view. Click the Target Console view to see the result of running the program:

```
Iris server started listening to port 7100
terminal_0: Listening for serial connection on port 5000
terminal_1: Listening for serial connection on port 5001
terminal_2: Listening for serial connection on port 5002
terminal_3: Listening for serial connection on port 5003
Iris server is reported on port 7100
Calling assembly function my_function with x0=4 and x1=5 results in 9
```

We can see from this that our assembly function is working, correctly adding the numbers 4 and 5 to give a result of 9.

5. Assembly language basics

This section of the guide introduces some of the basic concepts of assembly language programming, and explains how they are used when running a simple assembly program.

5.1 Registers

Most assembly instructions perform an operation on data in memory.

For example, we might want to do any of the following:

- Increment a number by 1
- Add two numbers together
- Subtract one number from another
- Multiply two numbers together, and then add the result to a third number

All data values must be stored in some kind of memory, otherwise the processor would forget all about them. When you think about computer memory, you might think about the hard drive in your laptop or the RAM memory chips. As far as the processor is concerned, these all provide long-term storage. But when we look at individual assembly code operations, data must be in a special kind of memory that is part of the processor itself: registers.

The Arm architecture provides 31 general-purpose registers.

For this guide, we will say that these registers are called x0 to x30, and they each contain 64 bits of data.



In fact, A64 assembly code lets you use these registers in two different ways. Each register can be used as a 64-bit x register (x0..x30), or as a 32-bit w register (w0..w30). These are two separate ways of looking at the same register. But for this guide, we just use the x0..x30 registers.

To read more about the general-purpose registers, see [Learn the architecture - A64 Instruction Set Architecture: Registers in AArch64 - general-purpose registers](#).

5.2 Loading data into registers

All data has to be in a register before it can be operated on by an Arm assembly instruction.

If this is the first time we have used a value, we will probably have to load it from main memory into a register. The `ldr` instruction transfers a single value between memory and the general-

purpose registers. For example, the following instruction loads 64 bits from <address> into register x0:

```
ldr    x0, [<address>]
```



For most A64 instructions, the first operand specifies the destination. So this instruction should be read as “Load TO x0 FROM <address>”.

This guide does not use the `ldr` instruction as often as you might expect because we are passing the data we need from our C program directly in registers.

For example, if you look at the example assembly code in [Calling an assembly function from C code](#), you can see we do not actually load any data into the registers before we start adding them together:

```
my_function:
    add    x0, x0, x1
    ret
```

This is because when we call the function `my_function` from our C code, the C code automatically puts the parameters in x0 and x1 for us. The rules that govern parameter passing and which registers are defined by a document called [Procedure Call Standard for the Arm 64-bit Architecture](#). This specification is complex, and beyond the scope of this guide. For now, it is enough to know that our registers are automatically loaded with values by the C program when we call the function, and if we changed the number or order of function parameters we might also need to change the registers used in the `add` instruction.

We can also move data around from one register to another to perform different operations. The `mov` instruction copies data from one register to another, as follows:

```
mov    x3, x0
```

This instruction copies the contents of register x0 to register x3.

Finally, we can also load registers with literal values. The following instruction loads register x2 with the integer value 7:

```
mov    x2, #7
```


5.3 Program flow

As with many other programming languages, Arm assembly language executes instructions one at a time, in order, until told to do something different. The examples in this guide change program flow using branch instructions, which jump to a different location in the program.

These branch locations are often specified using labels. You can see one such label in the example in [Calling an assembly function from C code](#):

```
my_function:
    add     x0, x0, x1
    ret
```

In this example, `my_function` is a label, and is used by the main C program to jump to our function.

The instruction `ret` also controls program flow. It tells the processor to return back to the calling function. In this case, we return to the function `main` in the C program.

We will look at how to control program flow in a later section, [Test instructions, branching, and loops](#), but for now it is enough to know that when reading an assembly code example you start at the top and read down, one instruction after another.

5.4 Arithmetic operations

Arm A64 assembly provides many different instructions that perform arithmetic operations. The format of each individual instruction dictates how many operands the instruction takes, and what they are used for.

Consider the example from [Calling an assembly function from C code](#):

```
my_function:
    add     x0, x0, x1
    ret
```

The `add` instruction in this example takes three operands as follows:

```
add <destination>, <register1>, <register2>
```

As we saw earlier, the first operand in most A64 instructions is the destination.

So, this instruction takes the value in register `x1`, adds it to the value in register `x0`, and finally stores the result in register `x0` overwriting the previous value that was stored in there.

5.5 A64 instructions

This section has introduced you to a small number of A64 instructions, such as `mov`, `add`, and `ret`.

However, there are many more A64 instructions available for you to use in your programs.

As you work through the examples in this guide, we will explain what each new A64 instruction does as we encounter it.

But when you come to write your own assembly programs, you will almost certainly want to know more about the other A64 instructions that are available, to find the exact instruction that suits your purpose.

[A64 base instructions](#) provides a complete list of all available A64 instructions for you to refer to.

6. Investigate registers with Arm Development Studio

Arm Development Studio includes Arm Debugger, a graphical debugger supporting software development on Arm processor-based targets and Fixed Virtual Platform (FVP) targets.

Arm Debugger provides a number of features that help us follow the execution of a program, see how it works, and examine what effect each instruction has on the processor's registers.

For example, Arm Debugger lets us do the following:

- Single-step through a program, stopping after every instruction.
- Inspect the values in the processor's registers to see how they have changed after each instruction.
- Set breakpoints to run to a certain point and then stop.

Let's use Arm Development Studio to investigate how our example program from [Calling an assembly function from C code](#) changes the values stored in different registers:

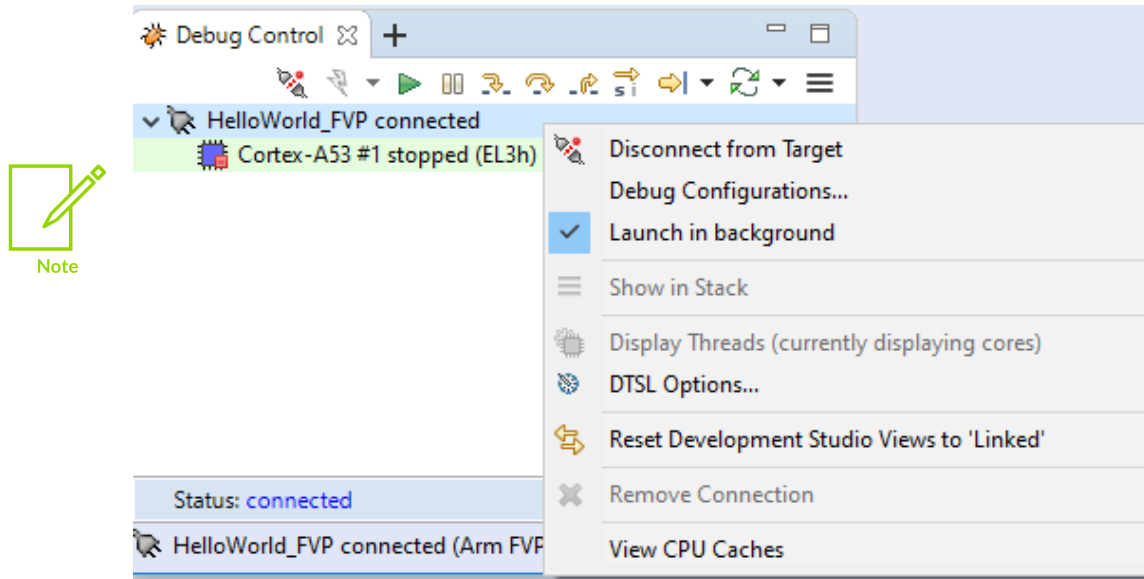
1. If you have not already done so, follow the instructions in [Calling an assembly function from C code](#) to get the HelloWorld project calling an assembly code function and running on an FVP.
2. Run your program on the FVP model.

In the Debug Control view, right-click HelloWorld_FVP and select Connect to Target.

The application loads on the target, and stops at the `main()` function, ready to run.

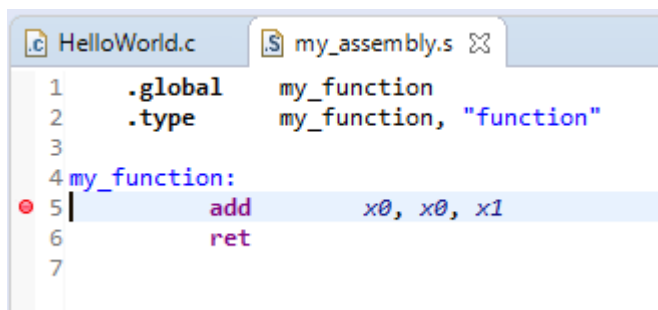
If the FVP is already running, disconnect it first by selecting Disconnect from Target.

Figure 6-1: Disconnect HelloWorld_FVP



3. Set a breakpoint on the `add` instruction in your assembly code by double-clicking in the left margin of the editor window next to the corresponding line of code. A red dot appears, indicating the breakpoint has been set, as shown in the following screenshot:

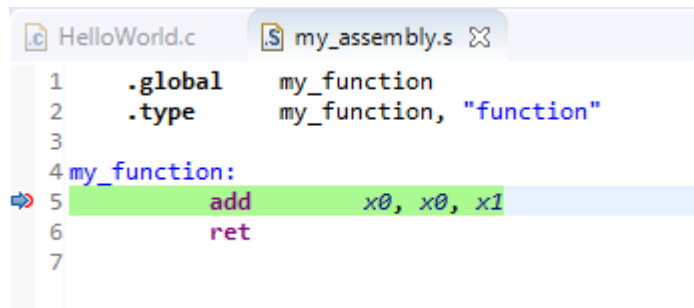
Figure 6-2: Set a breakpoint



A breakpoint specifies a location in your code where the debugger will pause execution. This will let us see what happens to the processor's registers when we execute the `add` instruction.

4. In the Debug Control view, click the green arrow icon to run the program up to the breakpoint you set.

Execution stops at the breakpoint, just before the `add` instruction runs. An arrow indicates the current instruction:

Figure 6-3: Run to the breakpoint

- Click the Registers tab, then expand the register tree as follows: AArch64 > Core to show the general-purpose registers.

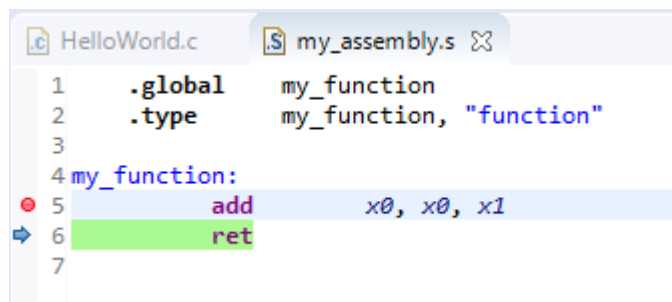
Figure 6-4: Registers before executing the add instruction

Name	Value	Size	Access
AArch64	589 of 589 registers		
Core	64 of 64 registers		
X0	0x0000000000000004	64	R/W
X1	0x0000000000000005	64	R/W
X2	0x0000000000000000	64	R/W

We can see that register x0 contains the value 4, and register x1 contains the value 5. These are the input values passed from the C code.

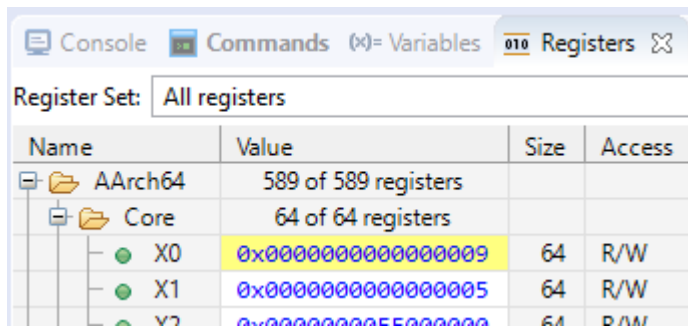
- In the Debug Control view, click the Step Source Line button  to run the next instruction, our add instruction.

The editor window updates to show that execution has moved to the next instruction:

Figure 6-5: Execution moves on by one instruction

- Look again at the Registers tab.

The x0 register's value has changed to 9, as a result of executing the add instruction:

Figure 6-6: Registers after executing the add instruction

Console Commands Variables Registers				
Register Set: All registers				
Name	Value	Size	Access	
AArch64	589 of 589 registers			
Core	64 of 64 registers			
X0	0x0000000000000009	64	R/W	
X1	0x0000000000000005	64	R/W	
V2	0x0000000055000000	64	R/W	

You can use this technique with all of the examples in this guide to see how individual instructions change the values in the processor's registers.

7. Test instructions, branching, and loops

We do not always want to have code that flows linearly. Sometimes we want to jump around our code, and have the program respond to conditions and inputs. To do this, we can use test and branch instructions.

Test instructions check whether a particular condition is met, and set condition flags based on the result in the Processor State (PSTATE). The condition flags are commonly referred to as the NZCV condition flags.

You can then respond to a flag being set, that is, a condition being met, by using branch instructions to tell your program to jump to different points in your code.

7.1 Test instructions

The `cmp` instruction lets you compare two values, for example to test if two values are equal, or if a value is less-than or greater-than a specified number.

The `cmp` instruction compares a value in a register with another number. This other number can either be in another register, or you can use an immediate value. For example:

```
cmp x0, x1    // Compare the number in register x0 with the number in register x1
cmp x0, #5    // Compare the number in register x0 with the number 5
```

If these numbers are the same, the instruction sets the Z (zero) flag in the NZCV condition flags.

If the second operand is larger than the first operand, the instruction sets the N (negative) flag in the NZCV condition flags.

The A64 ISA provides several other test instructions. There are different instruction variants for testing registers against other registers, or registers against immediate values. There are also instructions that let you test individual bits in a register. For the complete list of test instructions, see [A64 base instructions](#).

7.2 Test flags (NZCV)

As a result of the test instructions being executed, the [NZCV condition flags](#) are set. These flags can be used by branch instructions to determine where to go next in the program.

The NZCV condition flags include the following:

- **N** - Negative. The result of the operation was negative.
- **Z** - Zero. The result of the operation was Zero. Can occur when two values being compared are the same, or when you have decremented a value and the result is zero.

- **v** - Overflow (signed). The result of an arithmetic operation cannot fit in the available number of bits, and the MSB has been toggled. For A64 code using x registers, this flag is set when the result of the operation is greater than or equal to 2^{63} , or less than -2^{63} .
- **c** - Carry (unsigned). The result of the operation cannot fit in the available number of bits. For A64 code using x registers, this flag is set when the result of an operation is 65-bits long.

7.3 Branching and loops

Branching and loop instructions are closely related to test flags. Conditional branch instructions check the values in the test flags, and then jump to a certain point in the program based on the result.

When we put a conditional branch instruction at the end of a function, and then tell the branch instruction to jump to the start of the function until a flag is set, that is, until a condition is met, then we have ourselves a loop.

8. Example: decrement until equal

This example shows how a Branch if Not Equal `b.ne` instruction can be used inside a decrementing function, looping and testing until two numbers are the same.

Modify the example project you created in [Calling an assembly function from C code](#) as follows:

1. Use the following code for `HelloWorld.c`:

```
#include <stdio.h>

extern int my_function(int a, int b);

int main() {
    int a = 4;
    int b = 7;
    printf("After calling my_function, the values of both a and b are now %d\n",
        my_function(a, b));
    return(0);
}
```

2. Use the following code for `my_assembly.s`:

```
.global      my_function
.type       my_function, "function"
.p2align    4

my_function:
    sub x1, x1, #1
    sub x2, x0, x1
    cmp x2, #0
    b.ne my_function
    ret
```

3. Compile and run the project, and you see the following output:

```
After calling my_function, the values of both a and b are now 4
```

Let's go through the code line by line:

1. First, we run the C `main` function:

```
int a = 4;
int b = 7;
printf("After calling my_function, the values of both a and b are now %d\n",
    my_function(a, b));
```

When we call our assembly code, we pass two input values in registers: `x0` contains the value 4, and `x1` contains the value 7.

2. `my_function`:

This line defines a function entry point. The indented code is part of the function.

3. `sub x1, x1, #1`

Here we are decreasing the value in x1 by 1, making it 6, and then overwriting the original value in x1 with the new value.

4. `sub x2, x0, x1`

Here, we calculate the difference between x0 and x1, and store the difference in x2.

5. `cmp x2, #0`

Next, we are checking if the value in x2 is 0.

6. `b.ne my_function`

Finally, the `b.ne` instruction is called. This checks if the Z flag is set to 1. If it is set, then the function exits. Otherwise we jump back to the beginning of the function, defined by the function entry point, and repeat these steps.

In our example, the function will be looped three times, until x1 reaches 4. When x1 = 4, the result of the `cmp` instruction will set the Z flag. This time, when `b.ne` is called it will not jump back to the beginning of the loop. Instead it will go to the next line of code.

7. `ret`

Return to the calling function. In this case, the calling function is `main` in the C program.

The rules of the [Procedure Call Standard for the Arm 64-bit Architecture](#) dictate that our return value must be in register x0. In this example, the return value is the same value in x0 as at the start of the function. Our code does not modify the value in x0. This is now the value in both x0 and x1, because we have decremented x1 until both registers are the same.

9. Investigate NZCV flags with Arm Development Studio

In this section we will demonstrate how to view the NZCV condition flags in Arm Development Studio, using the example loop code from [Branching and loops](#).

If you have not already done so, complete the steps in [Before you begin](#) and [Calling an assembly function from C code](#) first. We are going to modify the project created by these steps.

1. From the Arm Development Studio IDE, open the `my_assembly.s` file.
2. Replace the code in `my_assembly.s` as follows:

```
.global      my_function
.type       my_function, "function"
.p2align    4

my_function:
    sub x1, x1, #1
    sub x2, x0, x1
    cmp x2, #0
    b.ne my_function
    ret
```

3. Open the `HelloWorld.c` file and replace the code as follows:

```
#include <stdio.h>

extern int my_function(int a, int b);

int main() {
    int a = 4;
    int b = 7;
    printf("After calling my_function, the values of both a and b are now %d\n",
        my_function(a, b));
    return(0);
}
```

4. Rebuild the project. Right-click on the HelloWorld project, and select Build Project.
5. Connect to the HelloWorld_FVP. Right-click on HelloWorld_FVP and select Connect to Target.
6. After the connection has established, we then need to configure the Registers view so that it only shows the registers we are interested in:
 - a. Open the Registers view. If this view is not showing, use the + button in the view menus to add the view to the IDE.
 - b. Click on the Register Set drop-down menu, and click Create.
 - c. In the Set Name field, enter `assembly101`.
 - d. Under All registers, expand AArch64 > Core and Add x0, x1, and x2.
 - e. Collapse Core and expand System > PSTATE and Add `nzcv`.
 - f. Click OK to save, and then expand the AArch64 sub-menus so that you can see all the specified registers.

7. Going back to the program, you will see that it has stopped at `main()`. Repeatedly press F5 to step through each line of the code, and watch as the register values update. When a register value changes as a result of a step, it is highlighted yellow.

In our example, our assembly function exits when the z flag is set.

Figure 9-1: Stepping through a program

The screenshot displays the Arm Development Studio interface. The top pane shows assembly code for a function named `my_function`. The code is as follows:

```

1  .global    my_function
2  .type      my_function, "function"
3 my_function:
4      sub x1, x1, #1
5      sub x2, x0, x1
6      cmp x2, #0
7  →  bne my_function
8      ret
9

```

The line `bne my_function` is highlighted in green, indicating the current instruction being executed. Below the code pane, the 'Registers' tab is selected. The 'Register Set' is 'assembly101'. The registers are organized into a tree structure:

- AArch64 (4 of 589 registers)
 - Core (3 of 64 registers)
 - X0: 0x0000000000000004 (64 bits, R/W)
 - X1: 0x0000000000000004 (64 bits, R/W)
 - X2: 0x0000000000000000 (64 bits, R/W)
 - System (1 of 365 registers)
 - PSTATE (1 of 5 registers)
 - NZCV: 0x60000000 (32 bits, R/W) - highlighted yellow
 - N: 0x0 (1 bit, R/W)
 - Z: 0x1 (1 bit, R/W) - highlighted yellow
 - C: 0x1 (1 bit, R/W)
 - V: 0x0 (1 bit, R/W)

10. Example: equation calculation

This example shows how to combine multiple instructions to perform more complex calculations, using registers to store interim results.

We will write a program to perform a common physics calculation, the [kinetic energy of an object](#).

An object's kinetic energy is the energy it possesses due to its motion. But don't worry too much about this: we are just using the calculation itself as an example. The equation is:

$$KE = 1/2 * mv^2$$

Where:

- KE is kinetic energy, in joules
- m is mass, in kilograms (kg)
- v is velocity, in meters per second (m/s)

That is, Kinetic Energy (KE) is half mass multiplied by velocity squared.

For example, an object with mass of 4kg moving at 5 m/s has kinetic energy 50 joules ($0.5 * 4 * (5 * 5)$).

We can not perform this calculation in a single step: it is too complex. We need to break it down into individual steps. These steps are:

1. Calculate the square of velocity
2. Multiply mass by the square of velocity
3. Divide the whole result by 2.

The following assembly code calculates the kinetic energy, given a mass passed to the function in register x0 and a velocity in x1.

```
my_function:           // On entry, x0 contains MASS and x1 contains VELOCITY
    mul    x2, x1, x1  // Calculate square of VELOCITY, put result in x2
    mul    x3, x2, x0  // Calculate MASS * VELOCITY SQUARED, put result in x3
    mov    x4, #2      // Put the value 2 in x4 to use in the next instruction
    udiv   x5, x3, x4   // Divide (MASS * VELOCITY SQUARED) by 2, put result in x5
    mov    x0, x5      // Move result back to x0 ready to return
    ret
```



Note

If you are wondering how we know that x0 contains mass and x1 velocity, rather than the other way around, it is because of the order that the values are passed.

Looking at the snippet of code in our C program that calls the assembly function: `my_function(a, b)` the first value (a) goes in register x0, and the second value (b) goes in register x1. If we had another argument, it would use x2, and so on.

More complex situations, such as very large data types or large numbers of parameters, pass values differently. But that is beyond the scope of this guide. See the [Procedure Call Standard for the Arm 64-bit Architecture](#) for more information.

This code does the following:

1. `mul x2, x1, x1`

The first thing we need to do is calculate the square of the velocity value, passed to our function in register x1. Your first thought might have been to look for an `sqr` assembly instruction or similar hoping to find a dedicated instruction to calculate the square of a number. If you looked, you will know that no such instruction exists. But we do not need one, because the square of a number is the result of multiplying that number by itself.

This line of code calculates the square of the value in x1 by multiplying it by itself, and storing the result in x2.

2. `mul x3, x2, x0`

Next, we need to take the result from the previous step and multiply it by the mass value, which is passed to our function in register x0. We store the result of this calculation in register x3.

Notice how we use registers to store these interim values. In this example, we are using different registers for each stage of the calculation. This is just for clarity. We could have been more efficient and re-used some of the registers. For example, we could have stored the result of this instruction in register x0.



Because the total number of registers in the processor is limited, the compiler has to work hard to decide the most efficient way to use the available registers. Bear this in mind if you read assembly code generated by a compiler. Heavy register re-use can make it more difficult for you to understand how a particular piece of code works.

3. `mov x4, #2`

This instruction puts the value 2 in register x4. This is needed in the next instruction, to divide by 2.

4. `udiv x5, x3, x4`

Now we need to divide the total we have calculated by 2. The `udiv` instruction performs division as follows:

```
udiv <dest>, <numerator>, <denominator>
```

That is, the `<numerator>` is divided by the `<denominator>`, and the result placed in `<dest>`.

So, our instruction divides the total of the calculation so far, in register x3, by the value of register x4, which is 2. The result is stored in x5.

5. `mov x0, x5`

Finally, we move the result of the last step to register x0 ready to return.

6. `ret`

Return from the function.

The rules of the [Procedure Call Standard for the Arm 64-bit Architecture](#) dictate that our return value must be in register x0. The previous instruction placed our result in x0, so we are ready to return.

If we modify our C program to use the values from our example at the start of this topic, then build and run, we should see the correct result.

Here is the C program with the correct input values:

```
int m = 4;
int v = 5;
printf("Calling assembly function my_function with x0=%d and x1=%d results
in %d\n", m, v, my_function(m, v));
```

And here is the result:

```
Calling assembly function my_function with x0=4 and x1=5 results in 50
```



Because this program uses integers, fractions in the result will be rounded down to the nearest integer. For example, if we used a mass of 5 and a velocity of 3, the result should be 22.5 ($0.5 \times 5 \times 9$). We can solve this by converting our code to use floating-point data, but that is beyond the scope of this guide.

11. Example: factorial calculation

This final example pulls together everything we have learned so far to tackle a more complex problem: calculating the [factorial](#) of a number.

The factorial of a number is defined as the product of all positive integers less than or equal to that number.

So, the factorial of 7 is calculated as $7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$, which equals 5040.

More generally, we can create an algorithm to calculate factorials as follows:

1. Start with the input number, for example 7.
2. Keep a running total, initially set to the input number.
3. Keep a counter, initially set to the input number.
4. Decrease the counter by 1.
5. If the counter is greater than zero, multiply the running total with the counter value, and loop back to step 4.
6. Otherwise, if the counter is zero, we have finished. The result is the value in our running total.

The following table shows how this algorithm computes the factorial of the number 7:

Counter	Running total
7	7
6	42 (7 x 6)
5	210 (42 x 5)
4	840 (210 x 4)
3	2520 (840 x 3)
2	5040 (2520 x 2)
1	5040 (5040 x 1)

The following assembly code calculates the factorial of a single number, passed to the function in register x0.

```
my_function:
    mov    x3, x0          // Copy register x0 to x3. Use x0 as running total, x3 as counter
loop:
    subs   x3, x3, #1      // Decrement counter in x3
    cmp    x3, #0          // Compare counter to zero
    b.eq   finish          // If result of comparison is "equal" (x3==0), branch to finish
    mul    x0, x0, x3       // Multiply running total and counter (x0*x3), then store in x0
    b      loop            // Branch to do another loop ; keep going until the counter is 0
finish:
    ret                    // Return the running total, which is already in x0
```

This code does the following:

1. `mov x3, x0`

Copy the initial input from register x0 to register x3. Remember that for most Arm instructions, the first operand is the destination. So this instruction can be read as “Move a value to register x3, from register x0”.

After executing this instruction, we have two copies of the input value. We can use one of these for the running total of our calculation, and the other as our counter. This code uses x0 as the running total, and x3 as the counter.

2. `loop:`

This label marks the start of our loop section. We will jump to this label repeatedly as the counter counts down to zero.

3. `subs x3, x3, #1`

Decrement the counter in x3.

The `subs` instruction means subtract, and uses the following format:

```
subs <dest>, <source>, <value>
```

We can read this instruction as “Subtract <value> from <source> and place the result in <dest>”.

For our instruction, `subs x3, x3, #1`, we can read this as “Subtract 1 from the current value in register x3, and store the result back into x3 overwriting the current value”. Which is just another way of saying “decrement the value in x3”.

4. `cmp x3, #0`

Compare the counter in register x3 to 0, and set the PSTATE Z (zero condition) flag to 1 if x3 is 0.

In fact, this instruction is not needed: we include it only to make it a little clearer how the program works. The previous `subs` instruction is a status setting subtract, as indicated by the final `s` on the name. So if the result of the subtract operation is zero, the `subs` command sets the Z flag to 1. Actually, the `cmp` instruction is really just an alias for the `subs` instruction.

5. `b.eq finish`

Check the result of the previous compare instruction, and if the counter is zero, branch to the end of the program.

The `b.eq` instruction checks whether the Z flag is 1. If it is, it branches to the specified label, `finish`. Otherwise, program execution continues as normal to the next instruction.

6. `mul x0, x0, x3`

Multiply the running total by the counter.

The `mul` instruction means multiply, and uses the following format:

```
mul <dest>, <op1>, <op2>
```

We can read this instruction as “Multiply <op1> and <op2> together, and place the result in <dest>”.

With our program, the counter is in x3, and the running total in x0. These values are multiplied together, and the result is then stored in register x0, overwriting the previous running total.

7. `b loop`

Branch back to the label `loop`. We keep looping until we reach the exit condition in step 5, branching out of the loop when the counter is zero.

8. `finish:`

This is the label used to jump out of the loop in step 5 and reach the end of the function.

9. `ret`

Return from the function.

The rules of the [Procedure Call Standard for the Arm 64-bit Architecture](#) dictate that our return value must be in register x0. The value we want to return is the latest value of our running total, and we have conveniently decided to use register x0 for this running total. So we do not need to do anything else: the result we want to return is already in x0, we just execute the `ret` instruction to exit the assembly function.

If we modify our C program to use the value 7 from our example at the start of this topic, then build and run, we should see the correct result.

Here is the C program with the required input value:

```
int a = 7;
printf("Calling assembly function my_function with x0=%d results in %d\n", a,
      my_function(a));
```

And here is the result:

```
Calling assembly function my_function with x0=7 results in 5040
```

12. Related information

Here are some resources related to material in this guide:

- Arm Development Studio resources:
 - [Download Arm Development Studio](#)
 - [Arm Development Studio Getting Started Guide](#)
- Wikipedia information related to examples:
 - [Factorial](#)
 - [Kinetic energy](#)
 - [Morse code](#)
- Arm reference information:
 - [A64 base instructions](#)
 - [AArch64 System Registers](#)

Arm Learn the Architecture guides:

- [Learn the architecture - A64 Instruction Set Architecture](#)
- [All Learn the Architecture guides](#)

Arm standards and specifications:

- [Procedure Call Standard for the Arm 64-bit Architecture](#)

13. Next steps

After completing this guide, you have seen several examples of how to write Arm A64 assembly code using a small subset of the A64 instruction set.

You can continue your learning by reading the [Learn the architecture - A64 Instruction Set Architecture](#) guide.

If you are ready to start writing your own A64 assembly code, you can start using more instructions. Consult the list of all available [A64 base instructions](#) to find out about new instructions you can use in your programs.

If you run into problems, you can visit the [Arm Community forums](#) to ask questions and get help. Alternatively, if you have an entitlement to Arm Support, you can [raise a support case](#).