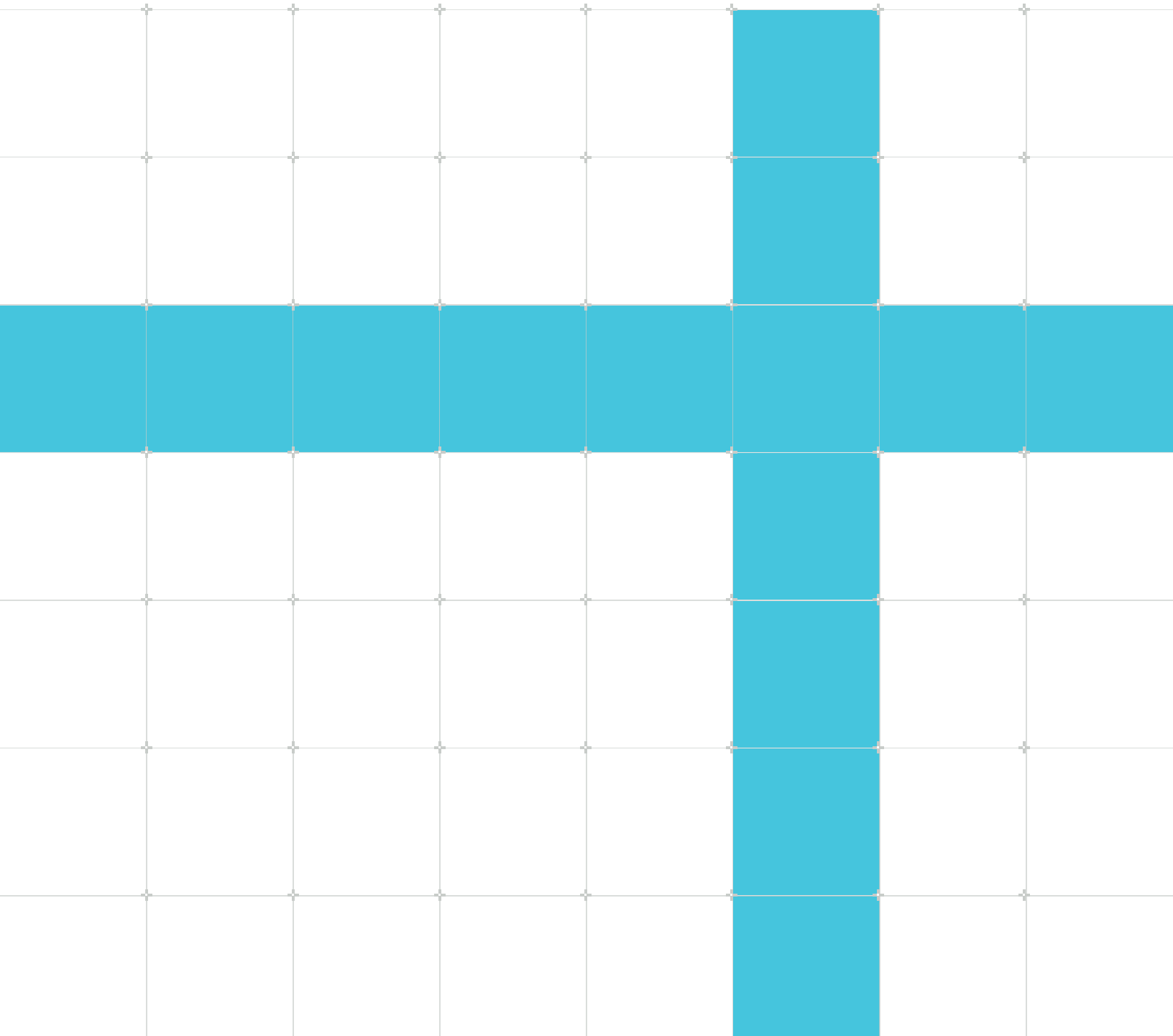
Getting Started with Arm Assembly Language

Version 2.0

Issue 02
107829_0200_02_en



Getting Started with Arm Assembly Language

Release information

Document history

Issue	Date	Confidentiality	Change
0100-02	21 December 2022	Non-Confidential	Initial release
0101-01	10 July 2023	Non-Confidential	Added another example
0200-02	7 September 2023	Non-Confidential	Added open source alternatives to Arm DS

Proprietary Notice

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

This guide introduces the basic concepts of Arm assembly language, shows you how to create and run assembly code, 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. The main program is written in C, and that program calls out to a function written in assembly code. This way, we can write our own assembly code function, without needing to worry about all the other code needed to make the program run.

We then use this method to look at different examples of simple programs written in Arm assembly language. We 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.

This guide focusses on writing assembly code to run on the Arm AArch64 Instruction Set Architecture (A64 ISA).

The A64 ISA is supported by Arm architectures such as Armv8-A and Armv9-A, which can be found in computers such as a Raspberry Pi. Alternatively, the architecture can be simulated using an Arm Fixed Virtual Platform (FVP) on a computer with a different ISA, such as an x86 computer.

This guide shows you how to run Arm assembly programs on both of these platforms.

2. Before you begin

To follow the examples in this guide, you need one of the following:

- A computer with the latest version of Arm Development Studio installed
- An x86_64 based computer, or virtual machine, running Linux
- An AArch64 based computer running Linux, for example a Raspberry Pi

You can check which architecture a Linux computer is using by running the following command in the terminal:

```
arch
```

The terminal outputs the architecture that the Operating System (OS) is running on. For example, it could be `x86_64` or `aarch64`.

The following sections give an overview of each platform, along with the setup process. Follow the instructions that apply to your platform before proceeding with the rest of the guide.

2.1 Arm Development Studio

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. It provides the best experience, combining all the required tools into one application. It also provides advanced debugging options, some of which are not available on other platforms.

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.

Complete 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.


Note

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.

2.2 Arm FVP on x86_64 Linux

As an alternative to Arm Development Studio, assembly programs can be compiled and run using a range of free and open-source tools that run on x86_64 Linux.

The code can be compiled using GNU Compiler Collection (GCC), and the program can run on an Arm Fixed Virtual Platform (FVP).


Note

This guide was written and tested using Ubuntu 22.04 LTS. Steps may vary slightly on other Linux distributions.

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

See Appendix: Hello World tutorial on x86_64 Linux for instructions on how to do this.

2.3 Native on AArch64 Linux

Instead of using a virtual model of an Arm processor, the assembly program can run natively on a processor supporting the A64 ISA.

The Armv8-A and Armv9-A architectures both support the A64 ISA. These can be found on many computers with an Arm processor, such as the Raspberry Pi 3 and the Raspberry Pi Zero 2 W.

The code is compiled directly on the AArch64 Linux computer using GNU Compiler Collection (GCC), and the program can be debugged using GNU Debugger (GDB).


Note

This guide was written and tested using a Raspberry Pi Zero 2 W, running Raspberry Pi OS (64-bit), kernel version 6.1. Steps may vary slightly on other Linux distributions, or on computers with a different Arm processor.

The computer must use a 64-bit OS. 32-bit OSs do not support A64 instructions.

Before you follow the examples in this guide, get a simple C program running on your AArch64 computer.

See Appendix: Hello World tutorial on native AArch64 Linux for instructions on how to do this.

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:

```
10010001000000000000010000000001
```

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 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.

First, we will look at the code, and then the rest of this section explores how to run it in each of the following three environments:

- Arm Development Studio
- An Arm FVP model on x86_64 Linux
- Natively on AArch64 Linux

A basic example of assembly code is shown below:

```
      .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 `.p2align` directive ensures that our code is properly aligned in memory to an 8-byte boundary.

We will take a look at how the rest of this assembly code works in Assembly language basics.

The C program to call this assembly program and print the output is as follows:

```
#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 in
 %d\n", a, b, my_function(a, b));
    return (0);
}
```

The `extern` keyword at the beginning of the program tells the compiler to look outside of the original program. In this case, it finds our assembly code, saved in a separate file, and runs it whenever we call the `my_function` command.

The following sections explain how to run this code on each of the three platforms.

4.1 Arm Development Studio

To modify the Hello World 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 assembly code shown in the previous page:

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

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 C 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
 in %d\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:

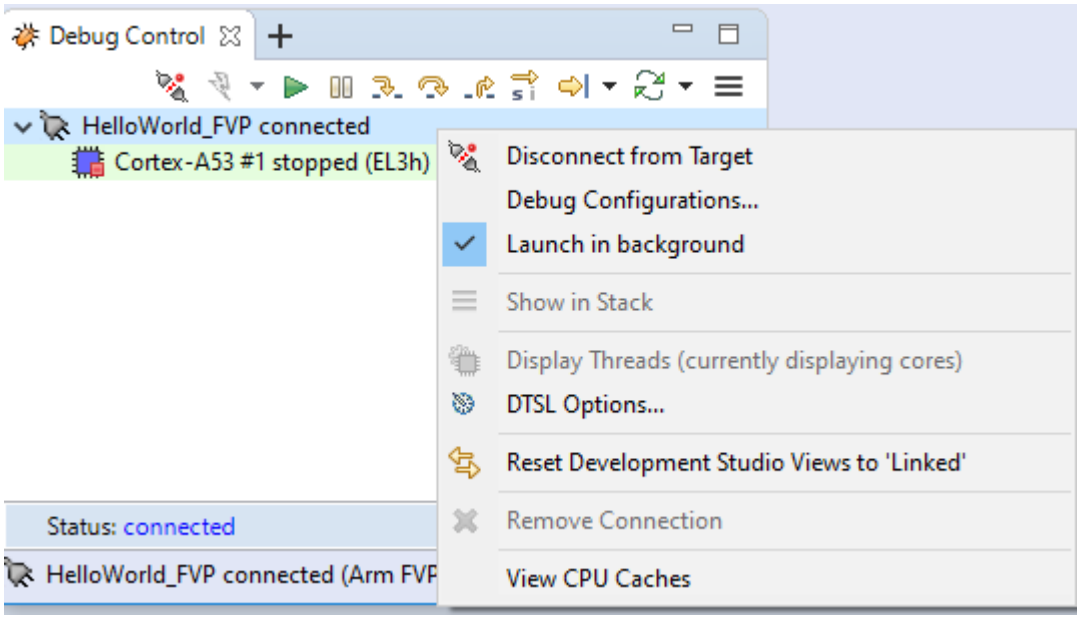
```
11:32:54 Build Finished. 0 errors, 0 warnings. (took 927ms)
```

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.

4.2 Arm FVP on x86_64 Linux

Modify the HelloWorld project, from Appendix: Hello World tutorial on x86_64 Linux, to call an assembly code function:

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

1. Create a new assembly code file.

   In your preferred text editor, create a file named `my_assembly.s` in the `src` folder.

   Enter the following assembly code in the `my_assembly.s` file:

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


Note

The cross-compiler, GCC, requires the end of the file to be a blank line, so make sure to include the last blank line in the assembly code.

2. Edit the `HelloWorld.c` file with the following C 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 in
 %d\n", a, b, my_function(a, b));
   return (0);
}
```

3. Build the project.

   Include the `my_assembly.s` file alongside the C file in the build command:

```
./arm-gnu-toolchain-12.3.rel1-x86_64-aarch64-none-elf/bin/aarch64-none-elf-gcc -
o out_a64 src/HelloWorld.c src/my_assembly.s -specs=arm-gnu-toolchain-12.3.rel1-
x86_64-aarch64-none-elf/aarch64-none-elf/lib/aem-ve.specs
```

4. Run the program on the FVP model using the same command as before:

```
./Foundation_Platform_11.21_15_Linux64/Foundation_Platformpkg/models/
Linux64_GCC-9.3/Foundation_Platform --image=out_a64
```

The terminal displays the result of the program:

```
user@Ubuntu:~/HelloWorld$ ./Foundation_Platform_11.21_15_Linux64/
Foundation_Platformpkg/models/Linux64_GCC-9.3/Foundation_Platform --image=out_a64

   Fast Models [11.21.15 (Mar 17 2023)]
   Copyright 2000-2023 ARM Limited.
   All Rights Reserved.

   WARNING: The AArch32 execution state can optionally be implemented at EL0, but
 is not implemented at EL1, EL2, or EL3 with Armv9.x
   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
   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.

4.3 Native on AArch64 Linux

Modify the HelloWorld project, from Appendix: Hello World tutorial on native AArch64 Linux, to call an assembly code function:

1. Create a new assembly code file.

   In your favorite text editor, create a file named `my_assembly.s` in the `src` folder.

   Enter the following assembly code in the `my_assembly.s` file:

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

2. Edit the `HelloWorld.c` file with the following C 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 in
 %d\n", a, b, my_function(a, b));
```

```
    return (0);
}
```

3. Build the project.

   Include the `my_assembly.s` file alongside the C file in the build command:

```
gcc -g -o out_a64 src/HelloWorld.c src/my_assembly.s
```

4. Run the program using the same command as before:

```
./out_a64
```

   The terminal displays the result of the program:

```
user@raspberrypi:~/HelloWorld $ ./out_a64
   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.

It explores:

- Registers, which are a type of memory that is built into the processor
- Loading data into registers from our assembly program
- Program flow, which is the order of assembly operations
- Basic arithmetic operations using the A64 ISA
- A64 base instructions, which are the full set of instructions available to use

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 A64 ISA 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.


Note

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 must be in a register before it an Arm assembly instruction can operate on it.

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>]
```


Note

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 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 Basic 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.

Other basic arithmetic operations function in the same way, such as the `sub` instruction to subtract, or the `mul` instruction to multiply. These instructions are explored more in the examples in this guide.

5.5 A64 base 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. Using a debugger to investigate registers

A debugger is a program used to find bugs, or errors, in the code.

The 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, the 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.

The debugger you use depends on which platform you are on:

- Arm Development Studio includes Arm Debugger, a graphical debugger supporting software development on Arm processor-based targets and FVP targets.
- On x86_64 Linux, it is not possible at current to debug assembly code running on the Arm FVP Foundation Package. Debugging requires the GDBRemoteConnection plug-in, which is only provided with the full Fast Models package.
- On AArch64 Linux, GDB is an open source debugger by GNU project which can be used to debug assembly code.

Debugging in Arm Development Studio and with GDB on AArch64 Linux is explored in the following sections.

6.1 Debugging with Arm Development Studio

Let's investigate how our example program from Calling an assembly function from C code changes the values stored in different registers:

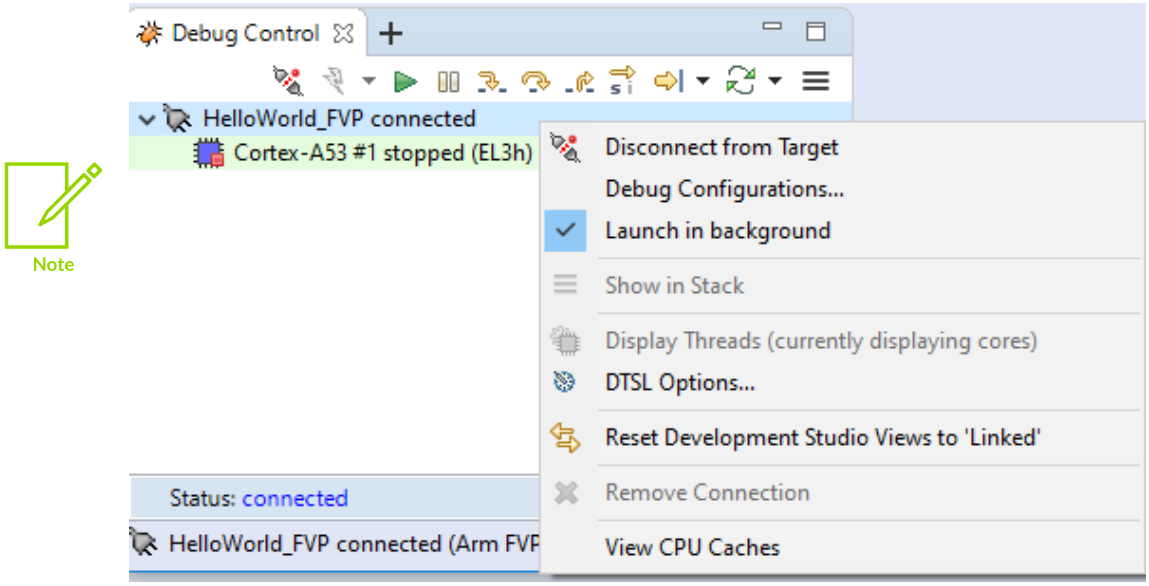
1. If you have not already done so, follow the instructions in Calling an assembly function from C code to get the HelloWorld project calling an assembly code function and running in Arm Development Studio.
2. Run your program on the FVP model.

   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.

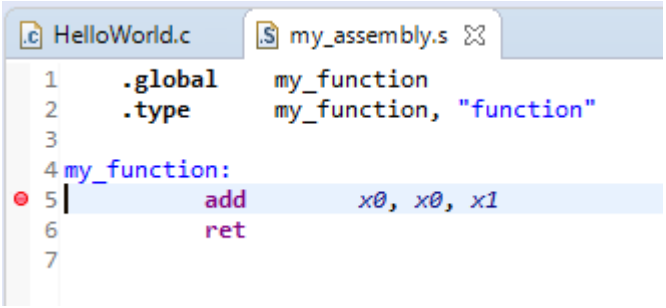
Figure 6-1: Disconnect HelloWorld_FVP



Note

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

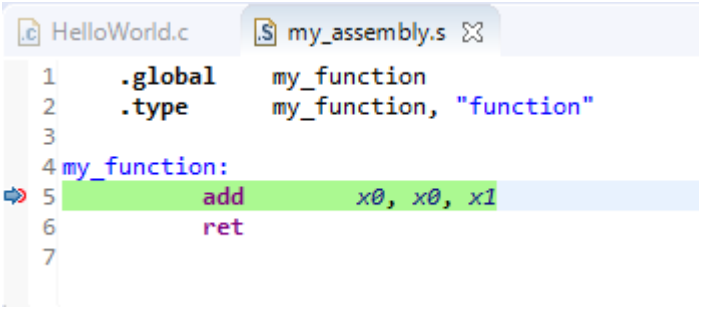


A breakpoint specifies a location in your code where the debugger will pause execution. This lets us see what happens to the processor's registers when we execute the `add` instruction.

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

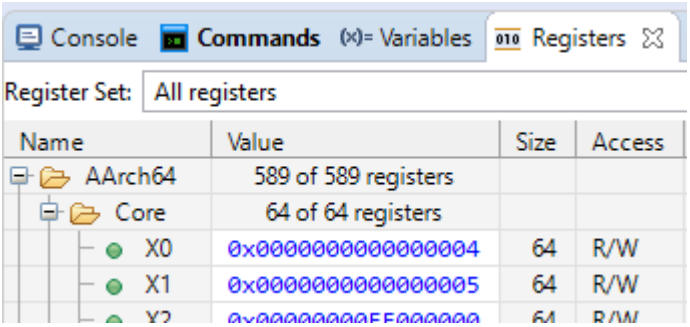
Execution stops at the breakpoint, just before the `add` instruction runs. An arrow indicates the current instruction:

Figure 6-3: Run to the breakpoint



5. 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

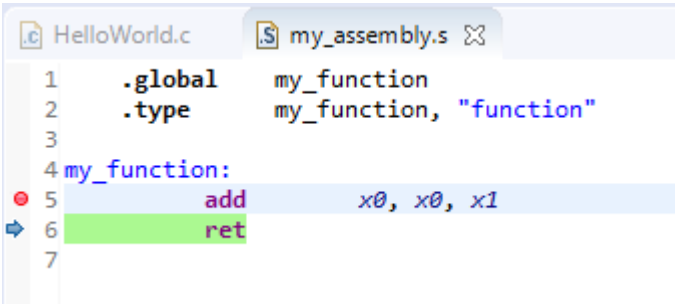


We can see that register x0 contains the value 4, and register x1 contains the value 5. These are the input values passed from the C code.

6. 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:

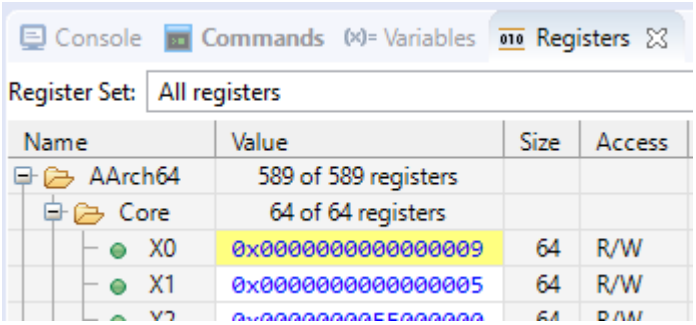
Figure 6-5: Execution moves on by one instruction



7. 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



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.

6.2 Debugging with GDB on AArch64 Linux

Let's 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 make a program that calls an assembly code function and runs natively on your AArch64 Linux computer.
2. Load the debugger, GDB, with the program by running the following command in the `arm_assembly` folder in the terminal:

```
gdb out_a64
```

3. Create a breakpoint at the `my_function` instruction in the assembly code with the following command:

```
(gdb) b my_function
```

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

Figure 6-7: Creating a breakpoint in the program

```
Reading symbols from out_a64...
(gdb) b my_function
Breakpoint 1 at 0x7c0: file my_assembly.s:6
```

4. Run the program up to the breakpoint with the following command:

```
(gdb) run
```

The terminal shows the program has paused on line 6 of `my_assembly.s`, and shows the next line to come in the code, which in this case is line 6, the `add` instruction.

Figure 6-8: Run to the breakpoint

```
(gdb) run
Starting program: /home/user/arm_assembly/out_a64

Breakpoint 1, my_function () at my_assembly.s:6
6               add     x0, x0, x1
```

5. Inspect the values stored in the registers with the following command:

```
(gdb) info r
```

This shows the values stored in all the registers at this point in the program.

To see only certain registers, specify the registers after the `info r` command. For example:

```
(gdb) info r x0 x1
```

shows just the x0 and x1 registers.

Figure 6-9: Registers before executing the add instruction

```
(gdb) info r x0 x1
x0             0x4                4
x1             0x5                5
```

The first column shows the name of the register. The second column shows the hexadecimal value stored in the register. The third column shows this value converted into decimal.

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.

6. Step through the code line by line using the command:

```
(gdb) s
```

After the first `s`, the `add` instruction is executed and GDB shows the instruction which comes next, in this case `ret`.

Before continuing, inspect the registers again with `info r x0 x1`. The x0 register's value has changed to 9, as a result of executing the `add` instruction:

Figure 6-10: Registers after executing the add instruction

```
(gdb) s
7               ret
(gdb) info r x0 x1
x0             0x9              9
x1             0x5              5
```

7. Now that the assembly function has reached the `ret` instruction, use the `c` command to Continue, and run the rest of the program continuously:

```
(gdb) c
```

8. Once the program has completed, use the `q` command to quit GDB and return to the terminal:

```
(gdb) q
```

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. This is done using test instructions 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.

- 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.
- 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, since one bit is used to store the sign, + or -, of the signed integer.

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.

The next example, Example: decrement until equal, explores a loop in more detail.

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 the `.c` file:

```
#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 the assembly `.s` file:

```
  .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. Build 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. 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:`

   Define a function entry point. The indented code is part of the function.

3. `sub x1, x1, #1`

Decrease the value in x1 by 1, making it 6, and then overwrite the original value in x1 with the new value.

4. `sub x2, x0, x1`

   Calculate the difference between x0 and x1, and store the difference in x2.

5. `cmp x2, #0`

   Check if the value in x2 is 0.

6. `b.ne my_function`

   Call the `b.ne` instruction. 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 loops back 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.

Now, we will demonstrate how to view the NZCV condition flags in each of the two debugging environments:

- Using the debugger in Arm Development Studio
- Using GDB on AArch64 Linux

8.1 Investigate NZCV flags with Arm Development Studio

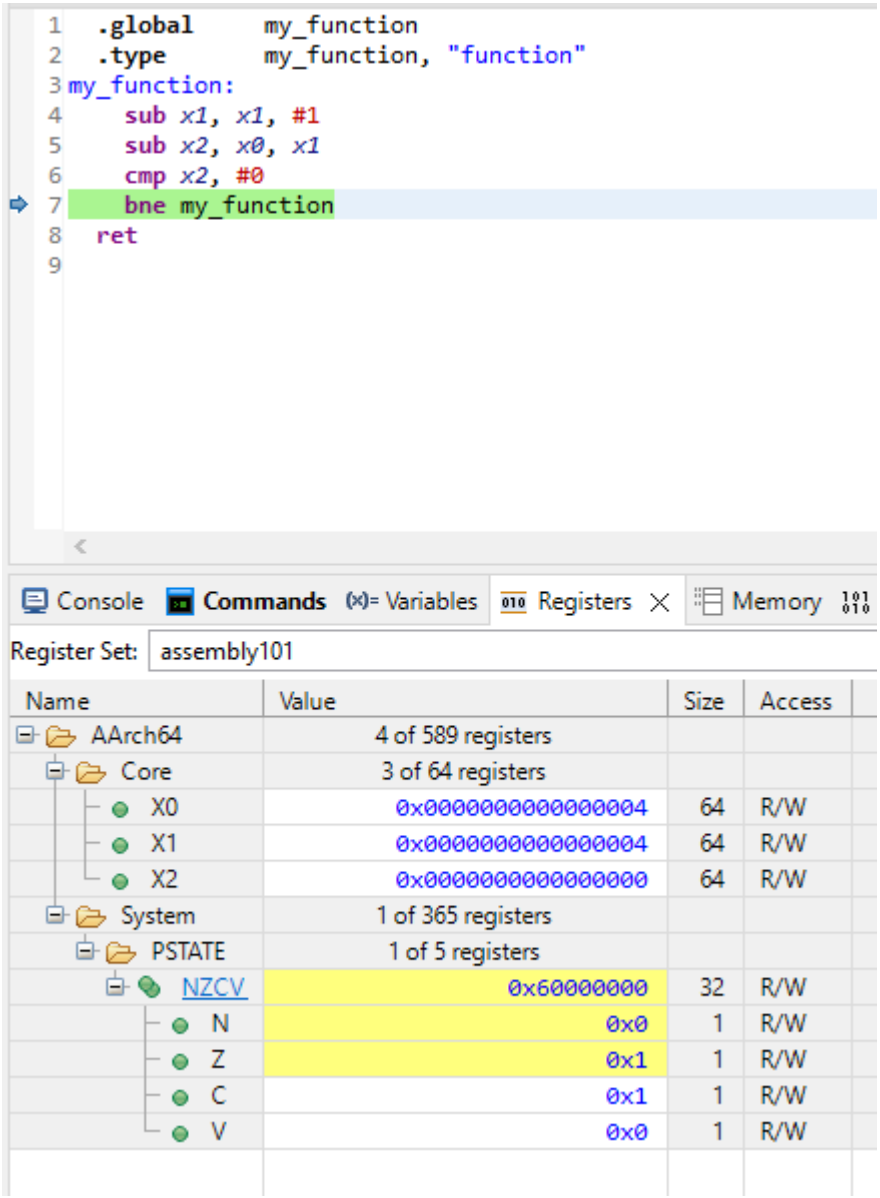
Arm Debugger, built into Arm Development Studio, allows us to inspect the values of the NZCV flags as the program runs.

1. Connect to the HelloWorld_FVP. Right-click on HelloWorld_FVP and select Connect to Target.
2. After the connection has established, 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.

3. 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 8-1: Stepping through a program



8.2 Investigate NZCV flags with GDB on AArch64 Linux

GDB allows us to inspect the values of the NZCV flags as the program runs.

1. From the terminal, load GDB with the compiled program:

```
gdb out_a64
```

2. Add a breakpoint at the start of `my_function`:

```
(gdb) b my_function
```

3. Start running the program:

```
(gdb) run
```

   The program runs until the breakpoint at the start of the assembly function.

4. Step through the program using `s` until the `cmp` instruction has been executed.

Figure 8-2: Stepping through the program line by line

```
Breakpoint 1, my_function () at my_assembly.s:6
6               sub     x1, x1, #1
(gdb) s
7               sub     x2, x0, x1
(gdb) s
8               cmp     x2, #0
(gdb) s
9               b.ne my_function
```

Remember that the line of assembly code shown after each `(gdb) s` is the next line of code to run, not the line that has just been executed. Once GDB prints the `cmp x2, #0` line, step once more to execute the compare.

5. Inspect the registers.

   Look at the value stored in x0, x1, x2 and the NZCV flags, which are stored in the `cpsr` register:

```
(gdb) info r x0 x1 x2 cpsr
```

   The values in x0, x1 and x2 are as expected, with x2 equal to x0 subtract x1. The third column of the `cpsr` register shows which flags are set.

Figure 8-3: Inspecting the registers and flags

```
(gdb) info r x0 x1 x2 cpsr
x0             0x4                 4
x1             0x6                 6
x2             0xfffffffffffffffe  -2
cpsr           0xa0200000          [ EL=0 SS C N ]
```

The `EL` and `SS` fields describe which state the processor is in. They are not important for this guide, see Process State for more information.

The `N` shows that the negative flag is set to 1, because x2 is less than zero in the `cmp` instruction.

6. Continue stepping through the program using `s`.

   Each time the `cmp` instruction is executed, check the values in the registers.

   When x1 is equal to x0, x2 is equal to 0. Now, `cmp` sets the `N` flag to 0, and the `Z` flag to 1. In GDB, this is shown in the third column, where `N` has been switched off and `Z` has now appeared.

Figure 8-4: The Zero flag is set

```
(gdb) s
9               b.ne my_function
(gdb) info r x0 x1 x2 cpsr
x0             0x4                 4
x1             0x4                 4
x2             0x0                 0
cpsr           0x60200000          [ EL=0 SS C Z ]
```

To get a better idea of what has changed, take a look at the hexadecimal values once converted to binary:

```
                    Hexadecimal:              Binary (32-bit):
N flag set:           a0200000          10100000001000000000000000000000
Z flag set:           60200000          01100000001000000000000000000000
                                        ↑↑↑↑
                                        NZCV
```

The most significant four bits of the `cpsr` register's binary value represent the `NZCV` flags respectively. Once `x2` is equal to `0`, the `N` bit is set to `0` and the `Z` bit is set to `1`.

Now that the `Z` flag is set, `b.ne` does not branch back to the start of the function, so our assembly function exits.

7. Run the rest of the program with `c`, and use `q` to quit GDB.

9. 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:

$$KE = 1/2 * m * v^2$$

Where:

- `KE` is kinetic energy, in joules (J)
- `m` is mass, in kilograms (kg)
- `v` is velocity, in meters per second (ms^{-1})

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

For example, an object with mass of 4 kg moving at 5 ms^{-1} has kinetic energy 50 J (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 function 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
```

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 <destination>, <numerator>, <denominator>
   ```

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

   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 put this code in our assembly function with 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:

```
#include <stdio.h>

extern int my_function(int m, int v);

int main()
{
    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 x 5 x 9). We can solve this by converting our code to use floating-point data, but that is beyond the scope of this guide.

10. Example: factorial calculation

This 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 × 6 × 5 × 4 × 3 × 2 × 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	7 × 6 = 42
5	42 × 5 = 210
4	210 × 4 = 840
3	840 × 3 = 2520
2	2520 × 2 = 5040
1	5040 × 1 = 5040

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 <destination>, <source>, <value>
   ```

   We can read this instruction as "Subtract `<value>` from `<source>` and place the result in `<destination>`".

   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 by 1".

4. `cmp  x3, #0`

   Compare the counter in register x3 to 0, which sets 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. 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 <destination>, <op1>, <op2>
   ```

We can read this instruction as "Multiply `<op1>` and `<op2>` together, and place the result in `<destination>`".

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:`

   The 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:

```
#include <stdio.h>

extern int my_function(int a);

int main()
{
    int a = 7;
    printf("Calling assembly function my_function with x0=%d results in %d\n", a,
 my_function(a));
    return (0);
}
```

And here is the result:

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

11. Example: checking prime numbers

This example combines several loops to test whether or not a number is prime.

A prime number is a positive integer, greater than 1, which is not the product of two smaller natural numbers. For example, 11 is a prime because it is not divisible by any integers other than 11 and 1. Conversely, 8 is not a prime because 8 = 2 × 4.

The following algorithm tests whether a given number is prime:

1. Start with the input number, for example 11.
2. Subtract 1 from the input number, this becomes our test number.
3. If the test number is equal to one, end the program because the input number is prime.
4. Calculate the remainder when dividing the input number by the test number. This is done by subtracting the test number from the input number until the result is smaller than the test number.
5. If the remainder is zero, end the program as the test number divides the input number, so it is not prime.
6. Decrease the test number by 1, then loop back to step 3.

The following table shows how the algorithm tests whether 11 is prime:

Test number	Check remainder
10	11 → 1: Remainder is greater than zero so does not divide
9	11 → 2: Remainder is greater than zero so does not divide
8	11 → 3: Remainder is greater than zero so does not divide
7	11 → 4: Remainder is greater than zero so does not divide
6	11 → 5: Remainder is greater than zero so does not divide
5	11 → 6 → 1: Remainder is greater than zero so does not divide
4	11 → 7 → 3: Remainder is greater than zero so does not divide
3	11 → 8 → 5 → 2: Remainder is greater than zero so does not divide
2	11 → 9 → 7 → 5 → 3 → 1: Remainder is greater than zero so does not divide
1	Program ends: 11 is prime

Similarly, this table shows how the algorithm tests whether 8 is prime

Test number	Check remainder
7	8 → 1: Remainder is greater than zero so does not divide
6	8 → 2: Remainder is greater than zero so does not divide
5	8 → 3: Remainder is greater than zero so does not divide
4	8 → 4 → 0: Remainder is zero, 4 divides 8 so it is not prime

Here is the corresponding Arm assembly code:

```
my_function:          // On entry, x0 contains the input number
    sub x1, x0, #1    // Calculate the first test number, which is the input
                      // number subtract 1

check:
    cmp x1, #1        // Compare the test number to one
    b.eq prime        // If the test number is 1, the input is prime so we branch
                      // to "prime"
    mov x2, x0        // Put the input number into a new register, ready for the
                      // repeated subtraction

subtract:
    sub x2, x2, x1    // Subtract the test number from the running total
    cmp x2, x1        // Compare the running total to the test number
    b.lt remainder    // If the running total is less than the test number, branch
                      // to the remainder test
    b subtract        // Otherwise, subtract the test number again

remainder:
    cmp x2, #0
    b.eq notprime     // If the remainder is zero, the test number divides the
                      //input so the input is not prime
    sub x1, x1, #1    // Otherwise, we move onto the next test number...
    b check           // ...and check again!

prime:
    mov x0, #1        // Use x0 as an indicator; 1 = prime
    ret

notprime:
    mov x0, #0        // Use x0 as an indicator; 0 = not prime
    ret
```

This code does the following:

1. `sub x1, x0, #1`

   Calculate the first test number, which is the input number subtract 1.

2. `check:`

   This section checks if the test number, x1, is 1. The code jumps back to this section each time we decrement the test number. If x1 ever reaches 1 then we know the input number is prime.

3. `cmp x1, #1`

   `b.eq prime`

   `mov x2, x0`

   Compare the register x1, the test number, to 1. The PSTATE Z flag is set to 1 if x1 is 1.

   If x1 is 1, the input number is prime, so branch to the section of the code that sets the output to prime.

   Otherwise, copy the input number to x2, where the repeated subtraction happens. Importantly, this leaves x0 unchanged so all the calculations are performed on x2, without forgetting the original input.

4. `subtract:`

   The code repeats this process until it can not subtract the test number any more, so it needs to loop back to this point.

5. ```
   sub x2, x2, x1

   cmp x2, x1

   b.lt remainder

   b subtract
   ```

   Decrement x2 by the test number, remembering that this means "Subtract x1 from x2, and store the result in x2".

   Compare x2 to the test number to see if another copy of the test number can be subtracted. If x2 is less than x1, the PSTATE N (negative condition) flag is set to 1.

   The `b.lt` instruction, `lt` for "Less than", checks the N flag: if it is 1, the first number in the compare is smaller than the second. In this case, it means the test number cannot be subtracted again, so branch to the `remainder` section to check whether the remainder is 0.

   Otherwise, subtract the test number again by branching back to the beginning of this section.

6. `remainder:`

   This section checks whether the remainder is zero.

7. ```
   cmp x2, #0

   b.eq notprime

   sub x1, x1, #1

   b check
   ```

   Compare the remainder, stored in x2, to zero. If they are equal, the PSTATE Z (zero condition) flag is set to 1.

   If the Z flag is 1, the test number divides the input number, so it is not prime. Branch to a section at the end to finish the program.

   Otherwise, subtract 1 from the test number, x1, to form the new test number. Branch back to the `check:` section to check if the test number is 1.

8. `prime:`

   This is the section that the program branches to if we establish that the input is prime.

9. ```
   mov x0, #1

   ret
   ```
   ```

If the number is prime, set the output (stored in x0) to 1, then exit the assembly function.

10. `notprime:`

    If a test number divides the input number, the program jumps to this section as the input is not prime.

11. `mov x0, #0`

    `ret`

    If the number is not prime, set the output to 0, then exit the assembly function.

Here is the C program, using the example input 11 from above:

```
#include <stdio.h>

extern int my_function(int a);

int main()
{
    int a = 11;
    printf("Calling assembly function my_function to test if %d is prime results in:
 %d (0 meaning not prime, 1 meaning prime)\n", a, my_function(a));
    return (0);
}
```

And the result:

```
Calling assembly function my_function to test if 11 is prime results in: 1 (0
 meaning not prime, 1 meaning prime)
```

11.1 Improving the program

Although this basic program works, there are a few issues with it, and a few ways to make it much faster.

11.1.1 Infinite loops

First, try running the program with the input number set to 1: `int a = 1;`.

Running on an FVP, nothing appears in the Target Console. Looking at the CLCD window, in around a minute over 1 billion instructions have been executed and the program has not terminated.

Figure 11-1: Stuck in an infinite loop

By definition, 1 is not a prime number, but the algorithm finds 1 - 1 = 0, and then start an infinite loop of subtracting 0 from 1.

This can be fixed by adding a `cmp` at the start of the program to make sure the input number is greater than 1.

11.1.2 Using unsigned division

Another way to check whether the test number divides the input number is by using the `udiv`, unsigned division, instruction.

However, the `udiv` instruction always returns an integer, truncating any fractional results. For example, calculating 11 ÷ 4 using `udiv` gives 2, instead of 2.75.

To check if the input number divides exactly by the test number, we need to test whether the division result has been truncated. This is done by multiplying the result by the test number. We can then compare this to the input number; if they are equal then the test number divides the input number exactly.

For example, to check if 3 divides 11:

Instruction	Result
`udiv`	11 ÷ 4 = 2 (truncated)
`mul`	2 × 4 = 8
`cmp`	8 is not equal to 11, so 4 does not divide 11 exactly

Whereas to check if 4 divides 8:

Instruction	Result
`udiv`	8 ÷ 4 = 2
`mul`	2 × 4 = 8
`cmp`	8 is equal to 8, so 4 does divide 8 exactly

For small inputs, the repeated subtraction used originally has fewer calculations to perform. However, for larger inputs, the test number needs to be subtracted hundreds of times, so the `udiv` algorithm will be significantly faster.

11.1.3 Reducing the number of calculations

The original algorithm uses a brute force method, checking every integer between 1 and the input number. Whilst it gives the correct result, it is not necessary to test every number.

Every factor of the input number must come in a pair. For example, 24 is divisible by 12, because 2 × 12 = 24. Similarly, 3 × 8 = 24, and 4 × 6 =24.

When one of the numbers in the pair is large, the other must be small, so there is no need to test large numbers, as long as we have tested enough small numbers. In fact, it suffices to test only up to the square root of the input number; if there is a factor greater than the square root, then its partner must be less than the square root.

The A64 base instruction set does not include a square root function. Instead, the easiest way to implement this is as follows:

1. Start with the test number set to 2.
2. Check whether the test number multiplied by itself is greater than the input number. If so, all factors have been checked, the input is prime and the program ends.
3. Check whether the test number divides the input number. If so, the input is not prime and the program ends.
4. Increment the test number by one and loop back to step 2.

Although this may seem more complicated than the original algorithm, it greatly reduces the number of calculations that must be performed.

For example, to check if 11 is prime:

Test number	Check	Result
2	Squared:	2 × 2 = 4: Less than 11, so continue
	Remainder:	11 → 9 → 7 → 5 → 3 → 1: Remainder is greater than zero so does not divide
3	Squared:	3 × 3 = 9: Less than 11, so continue
	Remainder:	11 → 8 → 5 → 2: Remainder is greater than zero so does not divide
4	Squared:	4 × 4 = 16: Greater than 11, so program ends, 11 is prime

This is even more useful when testing very large primes. For example, say we wanted to test whether 1,045,327 is prime. Our original algorithm would require us to check each of the 1,045,326 numbers less than this. However, the new algorithm will only need to test 1022 numbers, since 1023 × 1023 is greater than the input number.

Implementing this in Arm assembly can be done in a similar way to the original program. Perhaps a good exercise to practice the skills you have learnt!

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
- Linux applications:
  - GCC
  - VS Code
  - GNU Make
- Raspberry Pi Foundation:
  - Raspberry Pi
- Wikipedia information related to examples:
  - Factorial
  - Kinetic energy
  - Morse code
  - Prime numbers.
- 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.

14. Appendix: Hello World tutorial on x86_64 Linux

This 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. It goes alongside a video tutorial, which can be found here: Compiling for Arm FVPs.

Before continuing with the tutorial, update all the packages on the system. This is done with the following commands in the terminal:

```
sudo apt update
sudo apt upgrade
```

Now, use the following steps to download the tools and get a basic `Hello World!` program running on an Arm FVP.

14.1 Making a project directory

In a new terminal window, make a directory for the project files and navigate into the project directory with the following commands:

```
mkdir HelloWorld
cd HelloWorld
```

Create a subdirectory, `src`, to store the code files, using the following command:

```
mkdir src
```

14.2 Downloading the tools

Download the following tools:

- The GCC Toolchain for AArch64 targets

  The x86_64 computer, known as the “host”, has a different ISA to the processor that the program will run on, known as the “target”. Hence, a cross compiler must be used to compile the code, allowing the host to compile for a different target. The specific cross-compiler needed can be downloaded from the Arm website.

  In a browser, navigate to the Arm GNU Toolchain Downloads page. Under the `x86_64 Linux hosted cross compilers` section, download the `AArch64 bare-metal target (aarch64-none-elf)` toolchain.

  Follow the instructions in the release notes to extract the archive into the `HelloWorld` directory.

- The Arm Foundation FVP

  The FVP is a complete simulation of an Arm system, which we will run our compiled programs on. Again, this can be downloaded from the Arm website. Certain specialized FVPs are available to download for free, without licensing the full FVP library.

  In a browser, navigate to the Arm Fixed Virtual Platforms page. Under the `Arm Architecture FVPs` section, download the `Armv-A Foundation AEM FVP (x86 Linux)`. Extract the model to the `HelloWorld` directory.

14.3 Creating the code file

Now that all the tools are in place, create a C file for the Hello World project.

Using your favorite text editor, such as GNU Nano or Vim, create a file named `HelloWorld.c` in the `src` folder.

You could also use an Integrated Development Environment (IDE), such as Visual Studio Code, if you prefer to use a graphical interface.

Enter the following code in the `HelloWorld.c` file:

```
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return (0);
}
```

14.4 Building the project

To build the project, open a terminal and navigate to the `HelloWorld` directory. Run the following command to build the project:

```
./arm-gnu-toolchain-12.3.rel1-x86_64-aarch64-none-elf/bin/aarch64-none-elf-gcc -o
 out_a64 src/HelloWorld.c -specs=arm-gnu-toolchain-12.3.rel1-x86_64-aarch64-none-
elf/aarch64-none-elf/lib/aem-ve.specs
```

The syntax of this command is as follows:

`./arm-gnu-toolchain-12.3.rel1-x86_64-aarch64-none-elf/bin/aarch64-none-elf-gcc`

This runs the GCC compiler application, which is located within the sub-directories of our `HelloWorld` directory. If the GCC toolchain was extracted to a different location, change the file path here.

`-o out_a64`

This tells GCC to save the output to a file called `out_a64`.

src/HelloWorld.c

This is the source file that is being compiled, located in the `src` directory.

-specs=arm-gnu-toolchain-12.3.rel1-x86_64-aarch64-none-elf/aarch64-none-elf/lib/aem-ve.specs

This specifies parameters to use when building the project, such as the memory address. They are stored in a `.specs` file, which is included in the GCC Toolchain for Arm Cortex-A processors download. If the GCC toolchain was extracted to a different location, change the file path here.

An `out_a64` file should be created in the `HelloWorld` directory. To check, run the `ls` command to list all files in the directory.

14.5 Running the project on an Arm FVP

Now that the code is compiled into an executable file, the program can be run on the FVP. Use the following terminal command from the `HelloWorld` directory to run the FVP with the program:

```
./Foundation_Platform_11.21_15_Linux64/Foundation_Platformpkg/models/
Linux64_GCC-9.3/Foundation_Platform --image=out_a64
```

The syntax of this command is as follows:

./Foundation_Platform_11.21_15_Linux64/Foundation_Platformpkg/models/Linux64_GCC-9.3/Foundation_Platform

This runs the FVP, which is located within the extracted files in the `HelloWorld` directory.

--image=out_a64

This tells the FVP model to load the compiled program and run it.

The result of the program is displayed in the terminal pane:

```
user@Ubuntu:~/HelloWorld$ ./Foundation_Platform_11.21_15_Linux64/
Foundation_Platformpkg/models/Linux64_GCC-9.3/Foundation_Platform --image=out_a64

   Fast Models [11.21.15 (Mar 17 2023)]
   Copyright 2000-2023 ARM Limited.
   All Rights Reserved.

   WARNING: The AArch32 execution state can optionally be implemented at EL0, but is
 not implemented at EL1, EL2, or EL3 with Armv9.x
   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
   Hello World!
```

We can see from this that our program is working, printing the output as desired.

15. Appendix: Hello World tutorial on native AArch64 Linux

This tutorial provides a simple C code example program, and shows you how to compile and run it on an AArch64 Linux system, such as a Raspberry Pi.

Before continuing with the tutorial, update all the packages on the system. This is done with the following commands in the terminal:

```
sudo apt update
sudo apt upgrade
```

Now, use the following steps to get a basic `Hello World!` program running on the Arm processor.

15.1 Making a project directory

In a new terminal window, make a directory for the project files and navigate into the project directory with the following commands:

```
mkdir HelloWorld
cd HelloWorld
```

Create a subdirectory, `src`, to store the code files, using the following command:

```
mkdir src
```

15.2 Downloading the tools

GCC is a compiler used to turn the code into an executable file. On most Linux distributions, GCC is an included package. To check whether GCC is installed, run the following command in the terminal:

```
gcc --version
```

If the version is listed, GCC is installed on your system. Otherwise, run the following command to install it:

```
sudo apt install gcc
```

Another tool is needed to debug the code. This is explained in Debugging with GDB on AArch64 Linux, so check that GDB is installed.

GDB also comes pre-installed on most Linux distributions. To check whether it is installed, run the following command in the terminal:

```
gdb --version
```

If the version is listed, GDB is installed on your system. Otherwise, run the following command to install it:

```
sudo apt install gdb
```

15.3 Creating the code file

Now that all the tools are in place, create a C file for the Hello World project.

Using your favorite text editor, such as GNU Nano, create a file named `HelloWorld.c` in the `src` folder.

Enter the following code in the `HelloWorld.c` file:

```
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return (0);
}
```

15.4 Building the project

To build the project, run the following command in the terminal from the `HelloWorld` directory:

```
gcc -g -o out_a64 src/HelloWorld.c
```

The syntax of this command is as follows:

`gcc`
: This runs the GCC application, which is used to compile the code.

`-g`
: This tells GCC to include debugging information in the compiled program. Debugging options are explored in Using a debugger to investigate registers.

`-o out_a64`
: This tells GCC to save the output to a file called `out_a64`.

`src/HelloWorld.c`
: This is the source file that is being compiled, located in the `src` directory.

An `out_a64` file should be created in the `HelloWorld` directory. To check, run the `ls` command to list all files in the directory.

15.5 Running the project

Now that the code is compiled into an executable file, the program can be run. Use the following command in the terminal to run the program:

```
./out_a64
```

The result of the program is displayed in the terminal pane:

```
Hello World!
```

We can see from this that our program is working, printing the output as desired.