



# Learn the Architecture - Getting started with A64 exercises

Version 1.0

## Non-Confidential

Copyright © 2020 Arm Limited (or its affiliates).  
All rights reserved.

## Issue 01

102422\_0100\_01\_en



## Learn the Architecture - Getting started with A64 exercises

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

### Release information

#### Document history

Issue	Date	Confidentiality	Change
0100-01	21 February 2020	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, has undertaken no analysis to identify or understand the scope and content of, third party 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 subsidiaries) 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 © 2020 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)

## 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](mailto:terms@arm.com).

# Contents

1. Overview.....	6
2. Data processing and flow control.....	7
3. Accessing memory.....	14
4. System control.....	18
5. Example solutions.....	24
6. GAS syntax reference.....	28

# 1. Overview

The purpose of this set of exercises is to let you try out your knowledge of A64 assembler. It can help consolidate the knowledge that you have gained from other guides in our series, and can help you become familiar with the Arm development tools.

## Before you begin

This set of exercises assumes that you are familiar with the A64 instruction set. To learn more about the A64 instruction set, read our [Armv8-A Instruction Set Architecture](#) guide.

This set of exercises also assumes that you are familiar, in general, with embedded programming and the C language. The Arm tools that we use in the exercises use GAS syntax for assembler. If you are not familiar with GAS syntax for assembler, see [GAS syntax reference](#).

Detailed instruction and system register descriptions are not included in these exercises. To complete the exercises, refer the A64 and system register descriptions on Arm Developer.

[Example solutions](#) contained worked solutions to the exercises.

## Support files

Accompanying these exercises are a [set of support files](#). These files provide framework projects to get you started.

## Tools required

These exercises rely on the Arm Development Studio for compilation tools, debugging, and a simulation platform. If you do not already have a copy of Arm Development Studio, you can [download an evaluation copy](#).

You do not need any experience with Arm Development Studio IDE to complete the exercises. If you have used the tools before, you might want to skip the sections that explain the interface for new users.



Note

We wrote these exercises using Arm Development Studio 2019.0. If you are using a later version of the tools, some of the screenshots in these exercises may look different to what you see.

---

## 2. Data processing and flow control

In this first exercise, you will write a simple assembler function, which will then be called from C. A framework project is provided, so you only need to implement the function body. To complete the exercise, you will need to use data processing instructions, conditional operations, and a knowledge of the Procedure Call Standard (PCS).

Like all the exercises, there is more than one valid solution. This means that your answer may not match the suggested solution that is shown in [Example solutions](#).

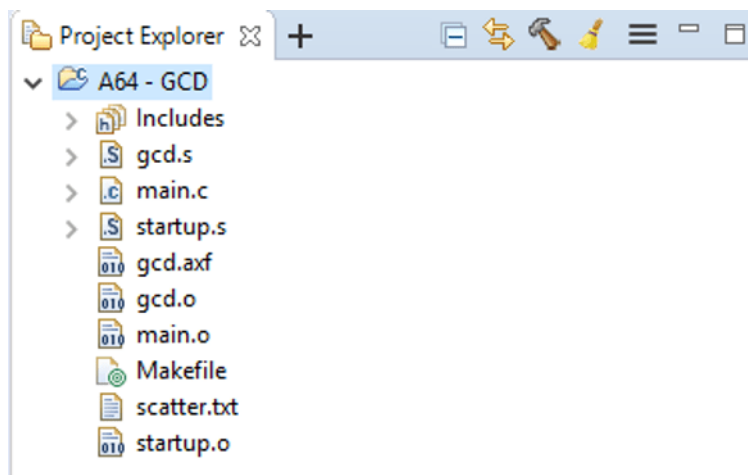
### Get started

First, load the provided framework project into Arm Development Studio, following these steps:

1. Launch the Arm Development Studio.
2. Click the Import Project icon or go to File -> Import.
3. Select General then Existing Projects into Workspace. Click Next.
4. Click Browse and navigate to where you downloaded the files that accompany the exercise. Select **1\_gcd**
5. Click Finish to import the project into Arm Development Studio.

The imported project then appears in the Project Explorer pane, as illustrated in this screenshot:

**Figure 2-1: View of the imported project in the project explorer**



You might need to expand the project to see the files.

Within the GCD project, you should see the following files:

`startup.s`

This is a simple reset handler. You will not need to modify this file for this exercise.

`main.c`

This contains the `C_main()` function, and implements a simple test harness for the function that you will develop.

`gcd.s`

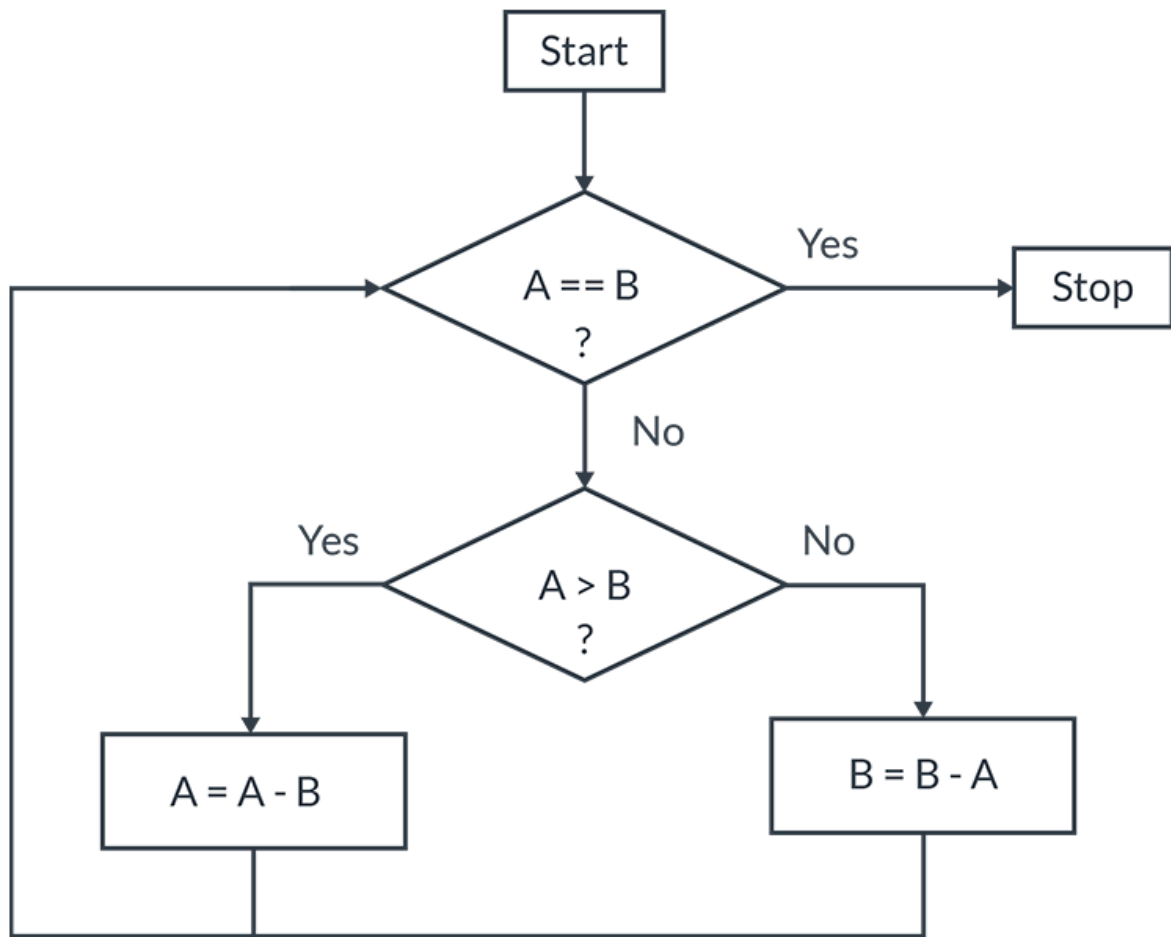
This is an A64 assembler file. This file contains an empty function definition that you will complete.

This exercise does not go into detail about the structure of the project or how it is built. If you are interested in how to construct an embedded image with Arm Compiler 6, see [Building your first embedded image](#).

### Implement an assembler function

In this exercise, you implement Euclid's algorithm for finding the greatest common denominator (GCD) of two integers. The algorithm is illustrated in the following flow chart:



**Figure 2-2: Algorithm flow chart**

To complete this exercise, follow these steps:

1. Open the file `gcd.s`. This file contains the outline for the GCD function, as you can see in the following code:

```
.global gcd
// uint32_t gcd(uint32_t a, uint32_t b)
.type gcd, @function
gcd:
//
//
// ADD YOUR CODE HERE
//
//
```

The function takes two 32-bit unsigned integers as arguments, `a` and `b`. The function returns a single 32-bit unsigned integer, which is the GCD of the two arguments.

2. Attempt to implement the function body using A64 assembler. You might want to read the sections Arithmetic and logic operations and Program flow in the [Armv8-A Instruction Set Architecture \(ISA\)](#)

Here are a few things for you to consider before getting started:

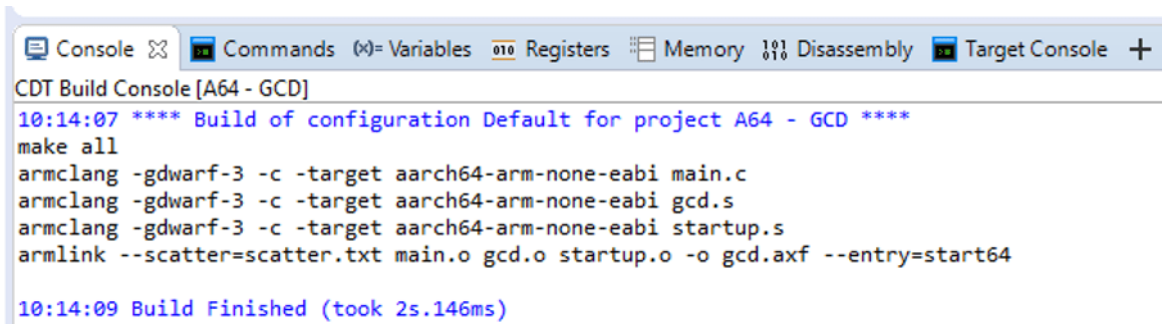
- Which registers will the arguments a and b be passed in?
- Given that 32-bit types are being used, what kind of general-purpose register should be used?
- Which register should the return value be in?
- How do you return from the function?

### Run the completed image

After you have completed the function, you can test it using the Fixed Virtual Platform (FVP) models that are provided with Arm Development Studio. Follow these steps:

3. Right-click on the project and select Build Project. Or, select Build Project from the Project menu. You will see the Console tab, which is near the bottom of the screen in a fresh installation. The Console tab shows the build messages. If the project builds successfully, the output will look like what you can see in the following screenshot:

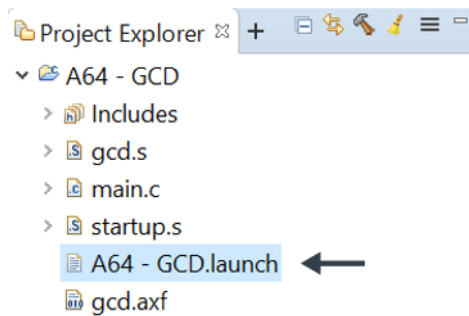
**Figure 2-3: Screenshot of console tab**



```
CDT Build Console [A64 - GCD]
10:14:07 **** Build of configuration Default for project A64 - GCD ****
make all
armclang -gdwarf-3 -c -target aarch64-arm-none-eabi main.c
armclang -gdwarf-3 -c -target aarch64-arm-none-eabi gcd.s
armclang -gdwarf-3 -c -target aarch64-arm-none-eabi startup.s
armlink --scatter=scatter.txt main.o gcd.o startup.o -o gcd.axf --entry=start64
10:14:09 Build Finished (took 2s.146ms)
```

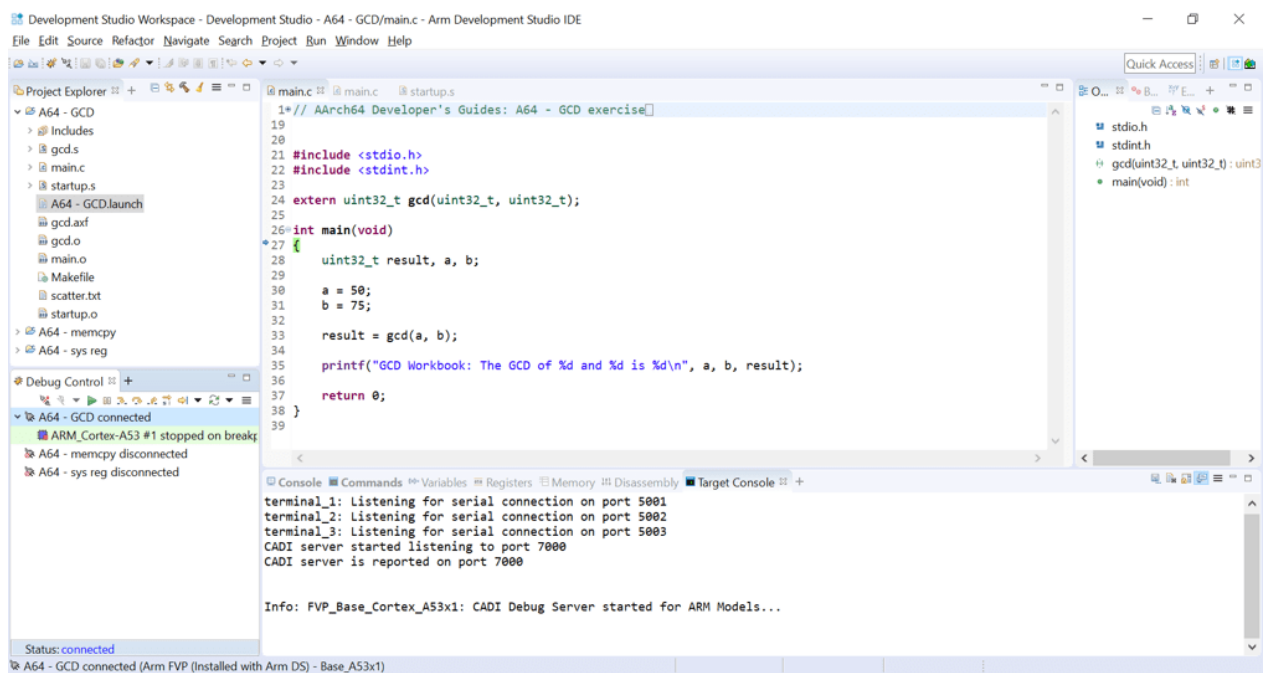
4. Check for any errors. If there are any, correct them and try to rebuild the project. Once you have successfully built your image, you can test it using the FVP models. This exercise uses an FVP with a single-core Cortex-A53 processor.

5. Double-click on the A64 - GCD.launcher file in the project, as shown in the following screenshot:

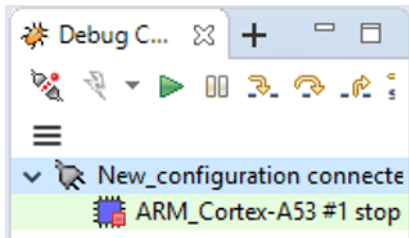
**Figure 2-4: Pointing to A64-GCD.launch in Project Explorer**

The Edit Configuration window opens.

6. Click Debug to launch the simulation. The debugger will launch the model, load the image, and run to the start of `main()`. You should see something that looks like the following screenshot:

**Figure 2-5: Screenshot of the debug simulation**

The icons in the Debug Control tab let you run, stop, or step the model, as shown here in this detail from the larger screenshot:

**Figure 2-6: Screenshot of debug controls**

7. Click the green arrow icon to run the model. The output from the simulator will be shown in the Target Console tab. The output for a successful run looks like what you can see in the following output log:

```
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
CADI server started listening to port 7000

Info: FVP_Base_Cortex_A53x1: CADI Debug Server started for ARM Models...
CADI server is reported on port 7000
GCD Workbook: The GCD of 50 and 75 is 25
```

## Debug your image



This section introduces the Arm Developer Studio controls for debugging. If you are new to the Arm Developer Studio, you can work through this section. If you have used Arm Developer Studio side before, or the DS-5 Debugger, you can skip this section.

To begin, we want a fresh simulation. Follow these steps:

1. Disconnect from the model using the Disconnect from Target in Debug Control pane.
2. Double-click on the entry for the model connection, to reconnect to the model.

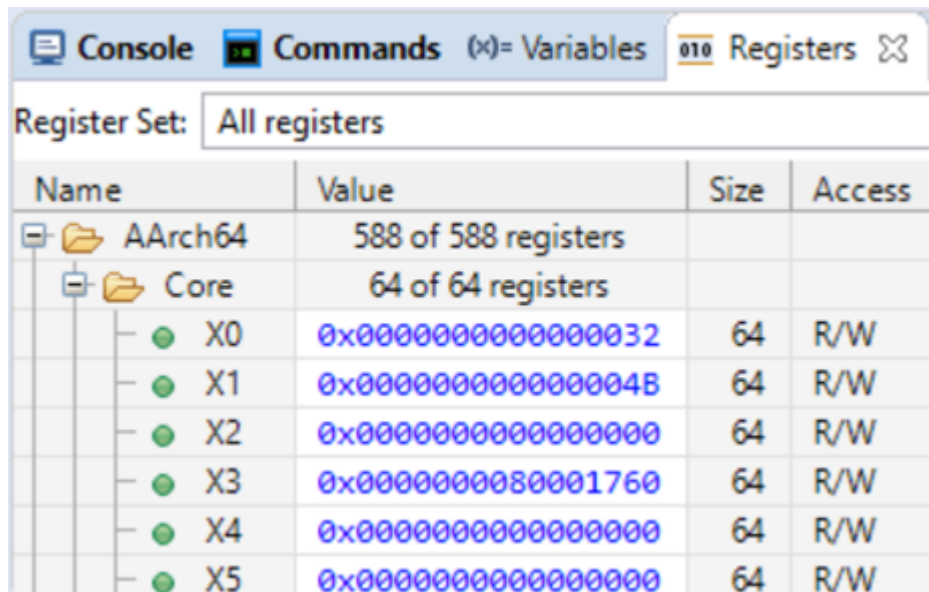
The connection is pre-configured to run the simulation to the start of `main()`.

At the start of the program, the Debug Control tab provides controls for running and stepping:

- Run - The simulation executes until it hits a breakpoint, or until it reaches the end of the program.
- Step Source Line - moves the simulation on either one C statement or one A64 instruction. The icon in Debug Control controls whether a C statement or A64 instruction is stepped.
- Step over or Step out are useful for functions. Step-over a function call will move execution to the next instruction after the function has returned. Step-out will move execution on until the current function has returned.

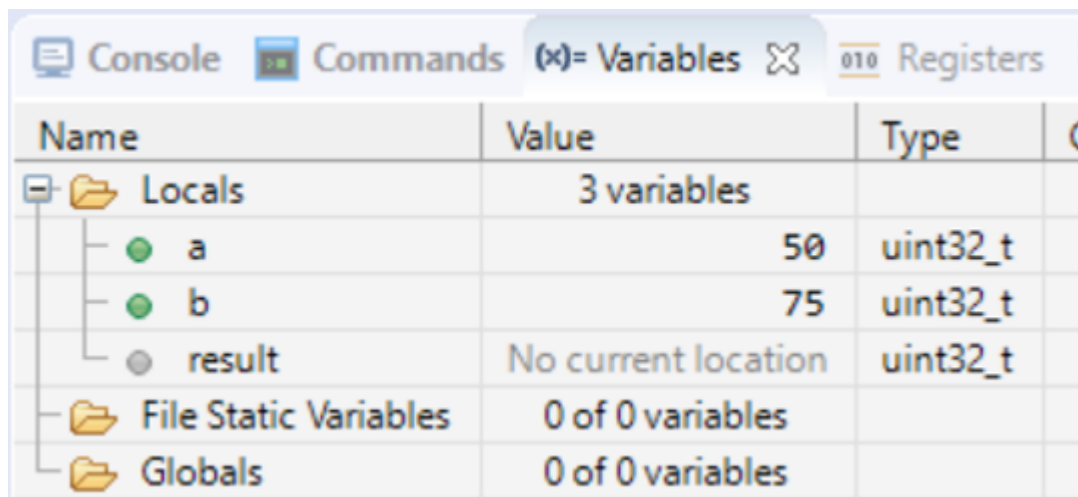
The current value of registers, or for C variables, can be found in the Registers and Variables pane, as you can see in this screenshot:

Figure 2-7: Screenshot of registers tabscreenshot of variables pane



Name	Value	Size	Access
AArch64	588 of 588 registers		
Core	64 of 64 registers		
X0	0x0000000000000032	64	R/W
X1	0x000000000000004B	64	R/W
X2	0x0000000000000000	64	R/W
X3	0x0000000080001760	64	R/W
X4	0x0000000000000000	64	R/W
X5	0x0000000000000000	64	R/W

Figure 2-8: Screenshot of registers tabscreenshot of variables pane



Name	Value	Type
Locals	3 variables	
a	50	uint32_t
b	75	uint32_t
result	No current location	uint32_t
File Static Variables	0 of 0 variables	
Globals	0 of 0 variables	

## 3. Accessing memory

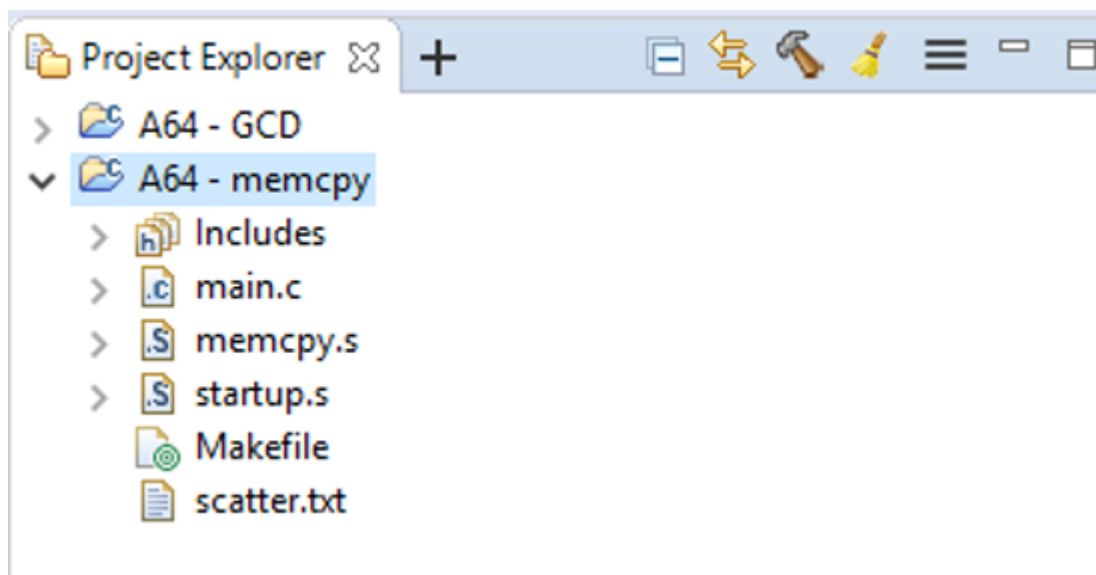
[Data processing and flow control](#) concentrated on data processing and flow control instructions. In this exercise, we show how to access memory with load and store instructions. To do this, we implement our own simple memory copy (`memcpy`) routine.

### Get started

Like with [Data processing and flow control](#), a framework project is provided to get you started. Follow these steps:

1. Import the 2\_memcpy project into the Arm Development Studio. The imported project then appears in the Project Explorer pane, as you can see in the following screenshot:

**Figure 3-1: Screenshot of project explorer**



You should see the following files in the memcpy project:

- `startup.s`

This is a simple reset handler. You will not need to modify this file for this exercise. Unlike Data processing and flow control, this startup file includes code to configure and enable the MMU.

- `main.c`

This contains the C `main()` function, and implements a simple test harness for the function that you will develop.

- `memcpy.s`

This is an A64 assembler file. This file contains an empty function definition that you will complete.

## Implement byte by byte copying

### 2. Open `memcpy.s`.

The following code shows the empty function that we are going to implement:

```
.global my_memcpy
// void my_memcpy(uint8_t* src, uint8_t* dst, uint32_t size_in_bytes)
.type my_memcpy, @function
my_memcpy:
//
//
//  ADD YOUR CODE HERE
//
//
RET
```

The function takes three arguments:

- `src` - a pointer to the source buffer, which points to first data item
- `dst` - a pointer to the destination buffer, which points to first empty location
- `size_in_bytes` - the number of bytes to be copied

For this exercise, we can assume that the pointers are to memory that is marked as Normal and that strict alignment checking is not enabled. This means that unaligned accesses are permitted. There are several possible approaches to implementing the function. We start with the simplest approach, which is a byte by byte copy. In pseudocode, we can represent this as you can see here:

```
while size_in_bytes greater than 0
    load byte from src
    increment src pointer by 1
    store byte to dst
    increment dst pointer by 1
    decrement size_in_bytes by 1
```

3. Implement the function, copying one byte at a time. Here are a few things to consider before getting started:

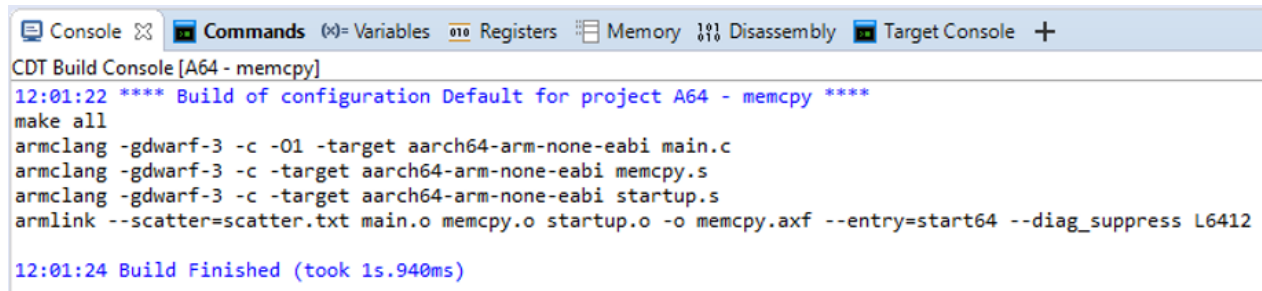
- What size are addresses in AArch64?
- How will you update the pointers after each iteration?
- What is the syntax for loading a sub-register sized quantity?

### Run the completed image

Once you have completed the function, you can test it using the Fixed Virtual Platform (FVP) models that are provided with Arm Development Studio.

4. Right-click on the project and select Build Project to build the project.

As in [Data processing and flow control](#), the Console tab shows the build messages. If the project builds successfully, the output will look like what you can see in this screenshot:

**Figure 3-2: Screenshot of console tab**


```

CDT Build Console [A64 - memcpy]
12:01:22 **** Build of configuration Default for project A64 - memcpy ****
make all
armclang -gdwarf-3 -c -O1 -target aarch64-arm-none-eabi main.c
armclang -gdwarf-3 -c -target aarch64-arm-none-eabi memcpy.s
armclang -gdwarf-3 -c -target aarch64-arm-none-eabi startup.s
armlink --scatter=scatter.txt main.o memcpy.o startup.o -o memcpy.axf --entry=start64 --diag_suppress L6412
12:01:24 Build Finished (took 1s.940ms)

```

5. Check for any errors. If there are any, correct them and try to rebuild the project. When you have successfully built your image, test it using the FVP models. This exercise uses an FVP with a single-core Cortex-A53 processor.

6. Launch the model using the `A64 - memcpy.launch` script in the project.

7. Click the green arrow icon to run the model. The Target Console tab shows the output from the simulator. The output for a successful run looks like this code:

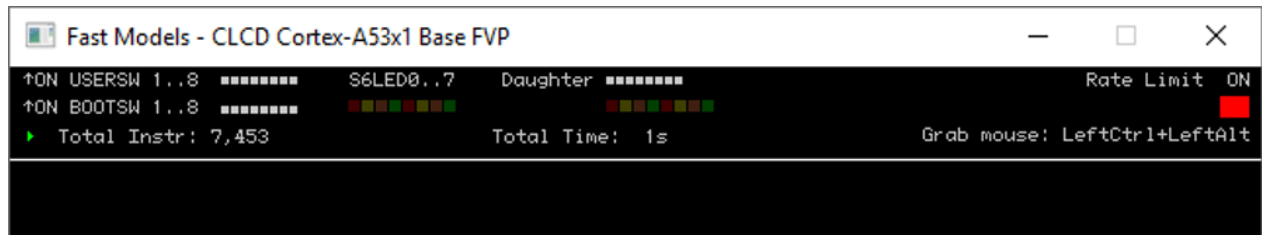
```

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
CADI server started listening to port 7000

Info: FVP_Base_Cortex_A53x1: CADI Debug Server started for ARM Models...
CADI server is reported on port 7000
Memcpy Workbook: Finished successfully

```

Every time you launch the model, you will see a window open, like the one that is shown here:

**Figure 3-3: Fast models simulation screen**

This window represents the LCD and switches of the simulated platform. These exercises do not use these features, but there is something else of interest. Total Instr reports the number of instructions that the simulator has executed since it was launched.

8. Make a note of the instruction count after running your implementation.





Note

Your figure might be different to that shown in the screenshot. The total instruction count 7,453 is based on the reference solution with Arm Compiler 6.12.

## Implement multi-byte copying

Copying one byte at a time is simple, but inefficient. For most copy operations, we want to transfer more than one byte at a time, so that we can reduce the number of iterations. We might also try to issue multiple loads and stores for each iteration. The next step is to modify `my_memcpy()` to use load and store pair instructions with X registers. This means that 128 bits, not 8 bits, are copied per iteration. The code needs also be able to handle data which is not a multiple of 128 bits in size. Follow these steps:

9. Modify the `my_memcpy()` function to use the LDP and STP instructions with X registers for the first iterations. Use smaller accesses for the last few bytes of the data.

10. Re-run the test program and check the instruction count. Has it changed? You should see that the instruction count has gone down. This screenshot shows the result using the reference solution:

**Figure 3-4: Fast models screenshot simulation**



Remember that this is the instruction count for the entire program, not just the running of `my_memcpy()`. But we can see that the more complex implementation has reduced the number of instructions needed to copy the data (7,453 instead of 6,778), at least with this size of buffer. The larger the data set, the bigger the reduction. However, with very small amounts of data, the new implementation might be slower.

11. Experiment with different sizes of data and different implementations of the copy routine. Consider using the wider floating-point registers.

## 4. System control

This exercise looks at the instructions for accessing system and special registers. System and special registers control the operation of the processor, for example cache configuration. Typically, system and special registers are programmed in start-up code, before switching to a C environment.

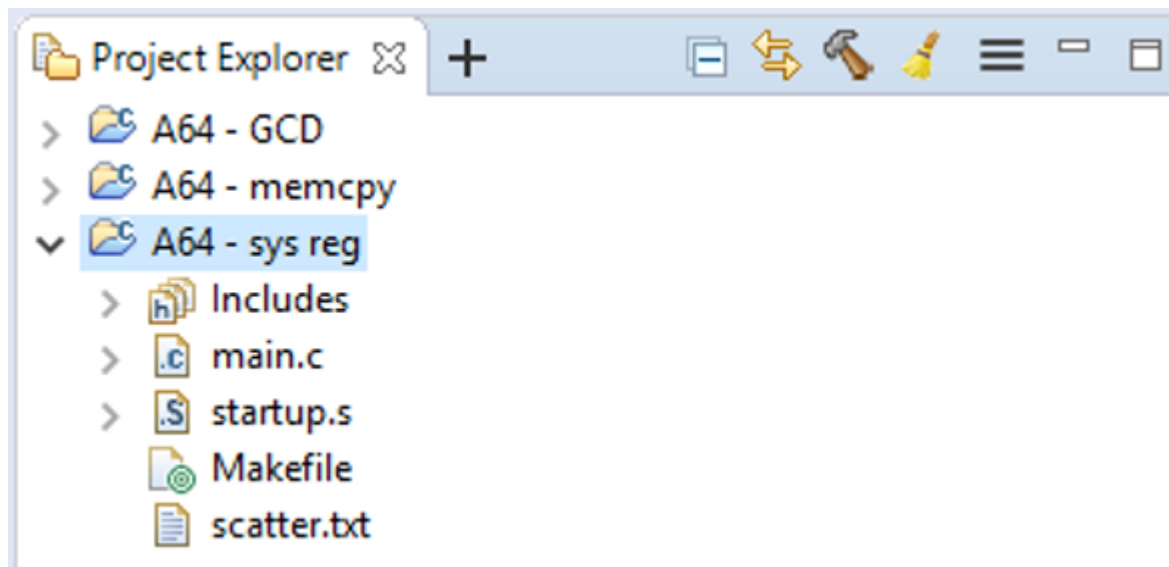
In [Data processing and flow control](#) and [Accessing memory](#) a `startup.s` file was provided, containing a very minimal reset handler. In this exercise, you will implement this file yourself. If you have a problem, look at the `startup.s` that is provided for the other examples as a reference.

### Get started

Like with [Accessing memory](#), a framework project is provided to get you started. Follow these steps:

1. Import the `3_sys_regs` project into the Arm Development Studio. The imported project then appears in the Project Explorer pane, as you can see in this screenshot:

**Figure 4-1: Screenshot of project explorer tab.**



Within the `**sys_reg**`

- `main.c`

A simple “hello world” C program.

- `startup.s`

This is the startup code that we will complete in the exercise.

## Implement the startup code

2. Open `startup.s`. The framework code is shown here:

```
.global start64
.type start64, @function
start64:

    // Check which core is running
    // -----
    // Core 0.0.0.0 should continue to execute
    // All other cores should be put into sleep (WFI)
    //
    // Your code here
    //

    // Disable trapping of CPTR_EL2 accesses or use of Adv.SIMD/FPU
    // -----
    //
    // Your code here
    //

    // Install EL3 vector table
    // -----
    //
    // Your code here
    //

    // The effect of changes to the system registers are
    // only guaranteed to be visible after a context
    // synchronization event. See the Barriers guide
    ISB

    // Branch to scatter loading and C library init code
    // -----
.global __main
B      __main
```

This example runs in EL3. For this exercise, we do not consider what would be necessary to switch Exception levels. There are three pieces of functionality that we need to implement:

- Detection of which core is being run on

We will run the example on a multiprocessor model, but the example is not written to be multi-threaded and needs to just run on core 0 (affinity 0.0.0.0). The startup code needs to check the ID of the core. If the ID of the core is not 0.0.0.0 then software should put the core to sleep using the WFI instruction.

- Clearing floating-point trap

The floating-point traps are Unknown at reset, but the C compiler assumes that floating-point operations are available before the C library initialization code (`__main`) is called. The startup code needs to ensure that the traps on accesses to the FPU are cleared.

- Install the EL3 vector table

Unlike earlier versions of the Arm architecture, in AArch64 there is no default vector table location. Software must install a vector table before the first exceptions are generated. This

example does not use exceptions itself, but it is good practice to install a simple vector table to capture unexpected exceptions.

## Floating point traps

The comment in `startup.s` tells us that the register that controls the FPU traps is `CPTR_EL3`. Let's start by looking at the [register description](#). The register contains several trap controls:

- `TCPAC` - Trap lower Exception level accesses to `CPTR_EL2` and `CPACR_EL1`
- `TAM` - Trap lower Exception level accesses to AMU
- `TTA` - Trap accesses to trace registers
- `TFP` - Trap EL3 use of FP registers

If you read the description of each field, you will find that a value of 0 means do not trap. Therefore, in this case we want to set the register to 0.

3. Write a sequence which will set `CPTR_EL3` to 0.

## Detect which core the software is running on

Each core has a unique affinity number, formatted as four 8-bit fields, as you can see here:

```
<aff3>.<aff2>.<aff1>.<aff0>
```

The affinity of a core can be read from the `MPIDR_EL1` register. Unlike [Data processing and flow control](#) and [Accessing memory](#), this exercise uses a model that contains multiple cores. However, the software is only written to run on one core. This means that the startup code must check which core it is running on, and if it is not core 0.0.0.0, the code should put the core to sleep using a `WFI`.

4. Implement code that reads `MPIDR_EL1` and check the affinity value, putting the core to sleep if not 0.0.0.0.

Things to consider:

- What is the format of the `MPIDR_EL1` register?
- How will you extract and compare the full affinity value?
- What will happen if the secondary cores are unexpectedly woken from standby?

## Install a vector table

The provided project includes a simple vector table at the end of `startup.s`. The format of the table, and how exceptions are handled more generally, is beyond the scope of these exercises. For more information, refer to [Exception model](#). For this exercise, we need to write the address of the vector table into the Vector Table Base Address register (`VBAR_EL3`). To do this, we need to know the address of the vector table. Let's look at the vector table in the project, which is shown in the following code:

```
.global vector_table
vector_table:
```

```
// -----
// Current EL with SP0
// -----
.balign 128
sync_current_el_sp0:
    B      .           //      Synchronous
    ...
```

The label `vector_table` marks the start of the table. We need to write the address of this label to `VBAR_EL3`. There are two pseudo instructions which allow you to get the address of a label:

- `ADR Xd, <label>`
- `LDR Xd, =<label>`

`ADR` only works for labels that are within the same compilation unit. `LDR` can also be used for imported global symbols.

5. Complete the code to set the EL3 vector table location, using either of these instructions.



There are two similar operations for `LDR`:

- `LDR Xd, <label>` Returns in `Xd` the value at `<label>`
- `LDR Xd, =<label>` Returns in `Xd` the address of `<label>`

## Run the completed image

When you have completed the function, you can test it using the Fixed Virtual Platform (FVP) models that are provided with Arm Development Studio.

6. Right-click on the project and select Build Project to build the project. Like in [Accessing memory](#), the Console tab shows the build messages. If the project builds successfully, the output will look like what you can see in the following screenshot:

**Figure 4-2: Screenshot of console tab**

```
CDT Build Console [A64 - sys reg]
15:41:19 **** Build of configuration Default for project A64 - sys reg ****
make all
armclang -gdwarf-3 -c -O1 -target aarch64-arm-none-eabi main.c
armclang -gdwarf-3 -c -target aarch64-arm-none-eabi startup.s
armlink --scatter=scatter.txt main.o startup.o -o hello_world.axf --entry=start64
15:41:21 Build Finished (took 2s.26ms)
```

7. Check for any errors. If there are any, correct them and try to rebuild the project.



If you look at the link command that is being issued for the image, it includes “`–entry=start64`”. This tells the compiler to set the entry point of the image to the label `start64`, which is the beginning of the startup code. The entry point is the address that the PC will be set to when the image is loaded into the simulation.

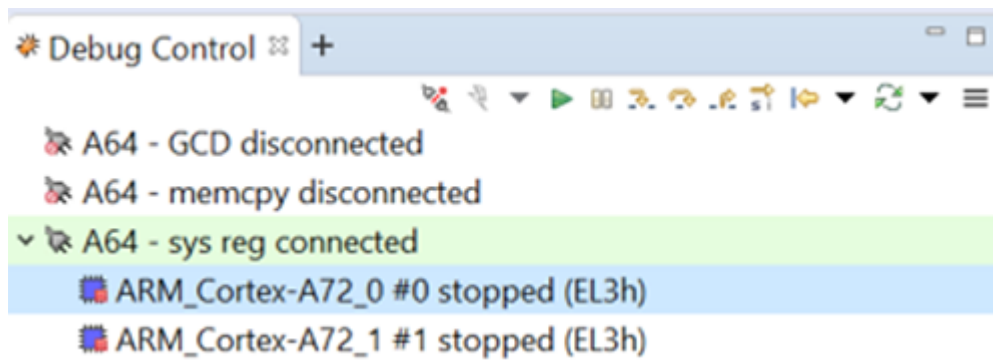
When you have successfully built your image, you can try it out. Follow these steps:

8. Use the `A64 - sys reg.launch` script in the project to launch the model. The model used for this exercise contains a dual-core Cortex-A72 processor and a quad-core Cortex-A53 processor. The affinity values for these cores are:

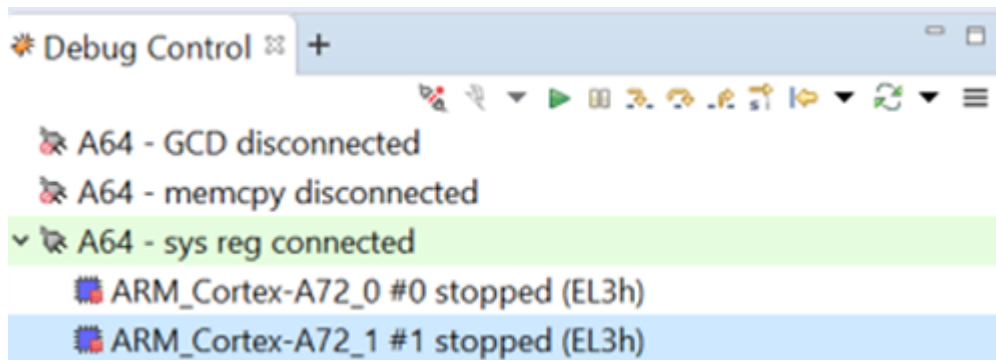
- 0.0.0.0 Cortex-A72, core 0
- 0.0.0.1 Cortex-A72, core 1
- 0.0.1.0 Cortex-A53, core 0
- 0.0.1.1 Cortex-A53, core 1
- 0.0.1.2 Cortex-A53, core 2
- 0.0.1.3 Cortex-A53, core 3

The debugger is configured to connect to the two Cortex-A72 cores, as you can see in the following screenshot:

**Figure 4-3: Screenshot of debug control**



The different debugger panes, like Source view and Register view, can only show information for one core at a time. The Debug Control pane selects which core's information is currently being displayed. In the preceding screenshot, you can see that core 0, `ARM_Cortex-A72_0`, is selected. If core 1 is selected, the Debug Control pane looks like what you can see in this screenshot:

**Figure 4-4: Screenshot of debug control**

Using the Register pane, we can manually check the `MPIDR_EL1` (AArch64 -> System -> ID) value that is reported by each core:

- ARM\_Cortex-A72\_0: 0x0000000080000000 -> affinity 0.0.0.0
- ARM\_Cortex-A72\_1: 0x0000000080000001 -> affinity 0.0.0.1

9. Step through the first few instructions of the image that is connected to core 0, to confirm that it is correctly checking the ID.

10. Disconnect and re-launch the model, this time stepping through the image on core 1 to compare the results. When you are satisfied that your code is working, run the image. The output from the simulator is shown in the Target Console tab. The output for a successful run looks like this code:

```
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
CADI server started listening to port 7000

Info: FVP_Base_Cortex_A72x2_A53x4: CADI Debug Server started for ARM Models...
CADI server is reported on port 7000
Hello world
```

The startup file for this example is basic, and a real image would perform more initialization. To explore this topic, see the [Bare Metal Boot](#) guide.

## 5. Example solutions

This section gives solutions to this set of exercises. There is more than one way of implementing the exercises, so your own solution might look different and still be correct.

### Data processing and flow control solution

The GCD algorithm that is shown in the flow chart in [Data processing and flow control](#) can be directly implemented in A64, as you can see in this code:

```
gcd:
    CMP    w0, w1          // Compare a and b
    B.EQ   end             // If they are equal, skip to the end
    B.LS   less_than       // If unsigned less than, branch to b = b - a
    SUB    w0, w0, w1       // a = a - b
    B      gcd             // Branch back to start
less_than:
    SUB    w1, w1, w0       // b = b - a
    B      gcd             // Branch back to start
end:
    RET
```

The code is using the LS condition, which equates to Unsigned lower or same. Alternatively, we could have checked for the Unsigned higher (HI) condition.

There are also Signed Greater Than (GT) and Signed Less Than (LS) conditions. These conditions are not used, because the code is treating the passed-in values as being unsigned. However, with the single set of test values that is used in the test program, you would get the same result.

Another way to implement the GCD algorithm is to use the conditional select instructions, as you can see in this code:

```
gcd:
    SUBS    w2, w0, w1       // tmp = a - b, with ALU flag update
    CSEL    w0, w2, w0, HI   // IF "unsigned higher" THEN a = tmp ELSE a = a
    CSNEG    w1, w1, w2, HI  // IF "unsigned higher" THEN b = b ELSE neg(tmp)
    B.NE    gcd             // Branch back to start
    RET
```

This solution is more efficient, because it uses fewer branches. Instead, the conditional select instructions are used to select the correct new value for a and b on each iteration.

### Access memory solution

For the byte by byte copy shown in [Accessing memory](#), here is a simple implementation:

```
my_memcpy:
    CBZ    w2, end          // Check for number of bytes being 0
    LDRB   w3, [x0], #1     // Load byte[n] from src, post-incrementing pointer
    STRB   w3, [x1], #1     // Store byte[n] to dst, post-incrementing pointer
    SUBS   w2, w2, #1       // Decrement number of bytes, updating ALU flags
    B.NE   my_memcpy        // Branch if number of bytes remaining not 0
```



As discussed in [Accessing memory](#), copying one byte at a time is inefficient. Here is a possible solution for multi-byte copying:

```
my_memcpy:
    // Loop until there is less than 16-bytes of data left
    CMP    w2, #15
    B.LS   my_memcpy_last_15_bytes
    LDP    x3, x4, [x0], #16
    STP    x3, x4, [x1], #16
    SUB    w2, w2, #16
    B      my_memcpy

    // Loop until there is less than 4-bytes of data left
my_memcpy_last_15_bytes:
    CMP    w2, #3
    B.LS   my_memcpy_last_3_bytes
    LDR    w3, [x0], #4
    STR    w3, [x1], #4
    SUB    w2, w2, #4
    B      my_memcpy_last_15_bytes

    // Copy the last remaining bytes (3 or fewer)
my_memcpy_last_3_bytes:
    CBZ    w2, my_memcpy_end
    LDRB   w3, [x0], #1
    STRB   w3, [x1], #1
    SUB    w2, w2, #1
    B      my_memcpy_last_3_bytes

my_memcpy_end:
    RET
```

This implementation improves on the previous code by using larger registers, and by copying multiple registers for each iteration. We could extend the code further by doing multiple `LDP` and `STP`s instructions per iteration and using the wider Q registers for the operations. In part, how we optimized the function might depend on what we expect the typical data size to be.

If you experiment with the C library `memcpy()`, you will see that the Arm-provided library provides multiple implementations. The compiler will attempt to select the one that is most appropriate to the context. For this exercise, modifying `main()` to call `memcpy()` results in the following code:

```
0x0000000080001804:    LDP    q0,q1,[x8,#0]
0x0000000080001808:    ADRP   x9,{pc}+0x4000 ; 0x80005808
0x000000008000180C:    ADD    x9,x9,#0x110
0x0000000080001810:    LDUR   q2,[x8,#0x4c]
0x0000000080001814:    LDP    q4,q3,[x8,#0x30]
0x0000000080001818:    STP    q0,q1,[x9,#0]
0x000000008000181C:    LDR    q1,[x8,#0x20]
0x0000000080001820:    MOV    w10,#0xbeef
0x0000000080001824:    MOV    x1,xzr
0x0000000080001828:    MOVK   w10,#0xdead,LSL #16
0x000000008000182C:    STUR   q2,[x9,#0x4c]
0x0000000080001830:    STP    q4,q3,[x9,#0x30]
0x0000000080001834:    STR    w10,[x8,#0x5c]
0x0000000080001838:    STR    q1,[x9,#0x20]
```



You can get the disassembly of an ELF image or object by double-clicking on the file within the Project Explorer tab and then selecting Disassembly.

In this instance, the compiler has optimized the output by fully inlining the code that is needed to perform the copy operation. The compiler could do this because the size of the copied data and the source and destination were both known at compile time.

This output was generated using Arm Compiler 6.12. The exact output for different compiler versions might vary.

## System control solution

The [System control](#) recreates the startup code that is used in [Data processing and flow control](#). Here is some code from the `startup.s` file that is provided with the GCD project:

```
.type start64, @function
start64:

    // Check which core is running
    // -----
    // Core 0.0.0.0 should continue to execute
    // All other cores should be put into sleep (WFI)
    MRS      x0, MPIDR_EL1
    UBFX     x1, x0, #32, #8      // Extract Aff3
    BFI      w0, w1, #24, #8      // Insert Aff3 into bits [31:24], so that [31:0]
                                // is now Aff3.Aff2.Aff1.Aff0
                                // Using w register means bits [63:32] are zeroed
    CBZ      w0, primary_core     // If 0.0.0.0, branch to code for primary core
1:
    WFI
    B        1b                  // If not 0.0.0.0, then go to sleep
```

Aff2, Aff1 and Aff0 are stored consecutively in bits [23:0]. However, Aff3 is stored in bits [39:32], with other fields in bits [31:24]. This register layout makes comparison more difficult. The code is extracting aff3 from the upper half of the register, then inserting it into bits [31:21] to make all the affinity fields consecutive.

The core with affinity 0.0.0.0 will branch to the `primary_core` label and continue with the rest of the start-up code. The other cores will execute the WFI instruction and go into Standby mode. If the cores are inadvertently woken from standby, there is a simple loop to capture them.

The code to initialize the floating-point traps is shown here:

```
// Disable trapping of CPTR_EL2 accesses or use of Adv.SIMD/FPU
// -----
MSR      CPTR_EL3, xzr          // Write 0, clearing all trap bits

// The effect of changes to the system registers are
// only guaranteed to be visible after a context
// synchronization event. See the Barriers guide
ISB
```

We want to clear all the trap bits to 0. The simplest way to do this is to write the zero-register, `xzr`, into the system register.

For installing the vector table, use this code:

```
// Install EL3 vector table
// -----
LDR    x0, =vector_table
MSR    VBAR_EL3, x0
```

The solution uses the `LDR` instruction, but `ADR` would also have worked.

## 6. GAS syntax reference

This set of exercises uses the GNU Assembler (GAS) syntax, which is the syntax that is required by Arm Compiler 6. A full description of the GAS syntax is beyond the scope of these exercises. This section briefly introduces the important pieces of syntax that are needed to complete the exercises.

Here is an example of a short assembler file containing a single function:

```
.section GCD,"ax"
.align 3

.global gcd
// uint32_t gcd(uint32_t a, uint32_t b)
.type gcd, @function
gcd:
//
//
//  ADD YOUR CODE HERE
//
//
RET
```

Going through the code line by line:

```
.section GCD, "ax"
```

This directive defines an ELF section, giving it the name `GCD` and marking it as executable ( `"ax"`). An ELF section is the smallest block of code and data that a compiler or linker can work on.

```
.align 3
```

The `align` directive sets the starting alignment of the code, in this case to  $2^3$  bytes.

```
.global gcd
```

The `global` directive can be used to either:

- Export a symbol that is defined within this file, making the symbol globally visible
- Import a symbol that will be defined somewhere else

In this example, the symbol `gcd` is defined in this file, therefore it is exporting the symbol.

```
.type gcd, @function
```

The `type` directive tells the tools what a symbol refers to. In this example we are saying that the symbol `gcd` refers to a function.

```
gcd:
```

This line defines a label called `gcd`. A colon (:) is needed after the name. This is different to the assembler syntax that used in the older Arm Compiler tools, for example, Arm Compiler 2.x, Arm Compiler 3.x, Arm Compiler 4.x, and Arm Compiler 5.x.

Looking at the makefile that is used in the template functions, the command to assemble or compile the source files is:

For C files:

```
armclang -gdwarf-3 -c -O1 -target=aarch64-arm-none-eabi <file>
```

For assembler files:

```
armclang -gdwarf-3 -c -target=aarch64-arm-none-eabi <file>
```

Taking the compiler arguments one at a time:

- `-gdwarf-3`

Tells the tool to generate debug data using the Dwarf-3 format. This is necessary to do source-level stepping in the debugger.

- `-c`

This tells the tool only to compile, or assemble, the file and not link. Linking is done as a separate step.

- `-O1`

For compiling C files, the level of optimization that is required. Level 1 is one of the lowest, meaning least optimized, which is useful for debugging these simple exercises.

- `-target=aarch64-arm-none-eabi`

This tells the tools the target architecture ABI, in this case Arm AArch64.

We could also have added:

- `-march=<version>`

This would let us specify which version of the architecture to target, for example `-march=armv8.1-a` means that that Armv8.1-A extensions are supported. The default is Armv8.0-A.