arm Education

Computer Architecture Course

LAB 1

Running Assembly Code on the Arm Education Core

Issue 1.0

Contents

1 Introduction

1.1 Lab overview

At the end of this lab, you will be able to:

- Create and modify an assembly source file that uses Armv8-A AArch64 instructions.
- Use the GCC compiler to assemble AArch64 instructions.
- Generate a hexadecimal Verilog memory model file from an ELF file to load onto the Arm Education Core.
- Compile and simulate Assembly code on a single-cycle processor (Arm Education Core) with a testbench file provided.
- Verify Assembly code using the Arm Education Core.
- Write a string copy operation using Armv8-A AArch64 instructions.

2 Requirements

Before attempting Lab 1, ensure that you have already completed the installation instructions in the *Getting Started Guide* provided with this course.

The prerequisites for this lab are:

- Familiarity with Arm assembly
- Verilog (2005)

Lab 1 exercise comes with supporting files provided in **Educore-SingleCycle.zip**.

3 What is the Arm Education Core?

Arm Education Core is an educational processor that implements a subset of the Armv8-A ISA. It is *one* implementation of the Armv8-A architecture and has limited functionality. It was developed purely for educational purposes and to demonstrate some computer architecture concepts. See the Introduction to Arm Education Core document provided in the Getting Started folder for more information. **Please refer to the *Arm Education Introduction to Computer Architecture Education Kit End User License Agreement* for terms of usage of Arm Education Core.**

In this lab, we will run a simple Arm Assembly code on the Arm Education Core. The Arm Education Core that is provided with this lab is a single-cycle processor, where an instruction is processed in a single clock cycle.

4 Coding in Arm Assembly

An Arm Assembly source file typically contains a sequence of statements. Each statement has three optional parts in the following order:

```
label       statement   // comment
```

For example:

```
mylabel     MOVZ X0, #1       // move 1 to register X0
            MOVZ X1, #2       // move 2 to register X1
```

- A `label` identifies the address of the instruction. It can be used as a target for branch instructions, or for certain load and store instructions. If the line has no label, a space or tab delimiter can be used to start the line.
- A `statement` can either be an assembly instruction, assembler directive, or a macro. An *assembler directive* tells the assembler to do certain things, and these directives do not generate machine language instructions. A macro is a text substitution representation.
- In the above example, the # symbol specifies the immediate value. An assembly instruction is encoded in terms of 1s and 0s, which a processor will have to decode to interpret the instruction. An immediate value is value or data stored as part of the instruction encoding itself. More information on instruction encodings will be covered in the next lab manual.
- The MOVZ instruction moves a 16-bit immediate value to a register. For more information, see MOVZ instruction in **Additional references**.

In this course lab, we will be introducing several assembly directives:

`.text`	Executable code is typically specified in the `.text` section.
`.global` *`symbol`*	Tells the assembler that the symbol or label is to be made globally visible to other source files and to the linker.
`.data`	Data for the code is typically placed in the `.data` section, which are used in the test cases for the final lab.

For more information on the Armv8-A A64 Instruction Set, see The A64 Instruction Set in **Additional references**.

5 String copy in Arm Assembly

In this exercise, we will write an Assembly source file that performs a string copy function. A C-program-equivalent function would be something like **strcpy(const char *src, char *dst)**. This function will load a character from memory, save it to the destination pointer, and increment both pointers until the end of the string.

For the exercises in the lab, we assume that:

- The source pointer ***src** is to be stored in register X0 and that it is pointing to address 0x0050.
- The destination pointer ***dst** is to be stored in register X1 and that it is pointing to address 0x013C.
- The characters “e” (ASCII 0x65) and “f” (ASCII 0x66) are stored in memory.

5.1 Exercise: Initializing content in memory

In the *Getting Started Guide*, we have created a workspace for the labs (**C:\Workspace**). We will now create the string copy Assembly source file.

1. In your working directory, create a folder called **Lab_1** (**C:\Workspace\Lab_1**).

2. In **Lab_1** folder, create an empty text file called **test_STRCPY.s** and type the following code in the created file:

```
.global _start
.text
_start:
```

Note: It is not a fixed rule that all label names must begin with an underscore. Underscores are typically used when it has to integrate with a higher level language like C, where some C compiler uses this convention to be able to be callable from a C code.

3. Under the **_start** label:
    a. Use the **MOVZ** instructions to initialize the source address (0x0050) in register X0 and destination address (0x013C) in register X1, respectively.

    b. Then, use the **MOVZ** instructions to shift value 0x65 in register X5 and value 0x66 in register X6.

4. Next, copy the following code to use the **STURB** instruction to load the values into memory:

```
STURB X5, [X0]
STURB X6, [X0, #1]
STURB WZR, [X0, #2]
```

Note:

- Arm Education Core has limited functionality and only supports a limited subset of the Armv8-A ISA. The Arm Education Core does not currently support the store instruction variant `STR <Xt>, [Xn]` as this instruction can be assembled in many ways of which the opcode is unsupported by the Arm Education Core. However, the Arm Education Core supports the **STUR** (Store Register Unscaled) instruction, which can give the same outcome. There are also specific **STUR** instructions depending on the data size to store, such as **STURB** and **STURH**.
- The difference between **STR** and **STUR** is that the **STUR** is the *Store (unscaled) Register* instruction that allows accessing 32/64-bit values that are not aligned to the size of the operand. This is unlike the STR operation, which requires the immediate offset to be in multiples of 4 or 8. However, **STR** and **STUR** instructions in this case gives the same result; therefore, we will be using **STUR** for the Arm Education Core instead.
- For more information on **STR** and **STUR** instructions, see the A64 Base Instructions in Additional references.

5. Save the file.

Answer the following question:

1. What does the **STURB** instruction do? How is **STURB** different from **STUR**? (**Hint: See the A64 Base Instructions in Additional references**)

2. What is **WZR** and how many bits wide is **WZR**? (Hint: Read about Zero register in Additional references)

3. What does instruction **STURB WZR, [X0, #2]** do?

4. What location address is the value of register X6 intended to be stored to?

5.2 Exercise: Assembling instructions and generate Verilog memory model file

To assemble **test_STRCPY.s** using the GNU C compiler (GCC) that is included in the GNU Toolchain for the A-profile Architecture, follow these steps:

1. Ensure that you are in the same directory as **test_STRCPY.s.**

2. Run the following command:

```
aarch64-none-elf-gcc -nostdlib -nodefaultlibs -lgcc -gdwarf-4
-Wa,-march=armv8-a -Wl,-Ttext=0x0 -Wl,-N -o test_STRCPY.elf
test_STRCPY.s
```

Note: In the *Getting Started Guide*, you have downloaded the GNU Toolchain for the A-profile Architecture that contains the GNU C Compiler (GCC). GCC recognizes the assembly file as an input and automatically invokes the GNU assembler and GNU linker. Running this command will generate an ELF file that contains the object code. The following table describes the switches used.

Command/switch	Description
aarch64-none-elf-gcc usage	**aarch64-none-elf-gcc [option switches] [input file]**
-nostdlib -nodefaultlibs -lgcc	Specify to not use standard system startup files or libraries when linking. Bypass libgcc.a, a library of internal subroutines GCC. Example of internal subroutine is __main. The -lgcc option ensures that you have no unresolved references to internal GCC library subroutines.
-gdwarf-4	Produce debugging information in DWARF version 4 format. DWARF is a debugging file format used by the compiler to support source-level debugging. It is also the format of debugging information within an object file.
-Wa,-march=armv8-a	Target architecture is Armv8-A; this setting is passed to the assembler with the -Wa switch.
-Wl, -Ttext=0x0	**-Ttext=0x0** specifies starting address (0x0) for the output file; this setting is passed to the linker with the -Wl switch.
-Wl, -N	-N sets the text and data sections to be readable and writable. Also do not page-align the data segment. This setting is passed to the linker with the -Wl switch.

-o	Specify the output filename

You would have observed the following error:

```
.s: Assembler messages:
.s: Warning: end of file not at end of a line; newline inserted
.s:11: Error: operand mismatch -- `sturb X5,[X0]'
.s:11: Info:    did you mean this?
.s:11: Info:        sturb w5, [x0]
.s:12: Error: operand mismatch -- `sturb X6,[X0,#1]'
.s:12: Info:    did you mean this?
.s:12: Info:        sturb w6, [x0, #1]
```

a. What is a W register and how does it differ from an X register? For example, how is register W5 different from register X5? (**Hint: Read the A64 General-Purpose Registers in Additional references**)

b. Why should the first operand STURB be a W register instead of an X register? (**Hint: See the A64 Base Instructions in Additional references**)

3. Fix the above error and rerun the command to assemble **test_STRCPY.s**. Ensure that there are no errors and an ELF file is generated.

4. What is the default endian setting for the GCC compiler if no specific endian switches are specified?

5.3 Task: Generate Verilog memory model file from ELF file

To convert the generated ELF file into encodings arranged in Verilog Memory Model format so that Arm Education Core can read it easily, follow these steps:

1. Ensure that you are in the same directory as **test_STRCPY.elf.**

2. Run the following command:

```
aarch64-none-elf-objcopy -O verilog test_STRCPY.elf test_STRCPY.mem
```

Command/switch	Description
aarch64-none-elf-objcopy	**aarch64-none-elf-objcopy [option switches] [input file] [output file]**
-O verilog	Set output target as Verilog Memory Model hexadecimal format, which is suitable for loading memory models of HDL simulators.

Observation: The contents of the generated .mem file should look like this:

```
@00000000
00 0A 80 D2 81 27 80 D2 A5 0C 80 D2 C6 0C 80 D2
05 00 00 38 06 10 00 38 1F 20 00 38
```

We will take a closer look at the meaning of these encodings in the next lab manual.

5.4 Testbench for Arm Education Core

Arm Education Core comes along with a testbench Verilog file called **test_Educore.v**. This testbench file instantiates the processor core module named **Educore** and defines the unified data and instruction memory called **memory**, as shown in Figure 1.

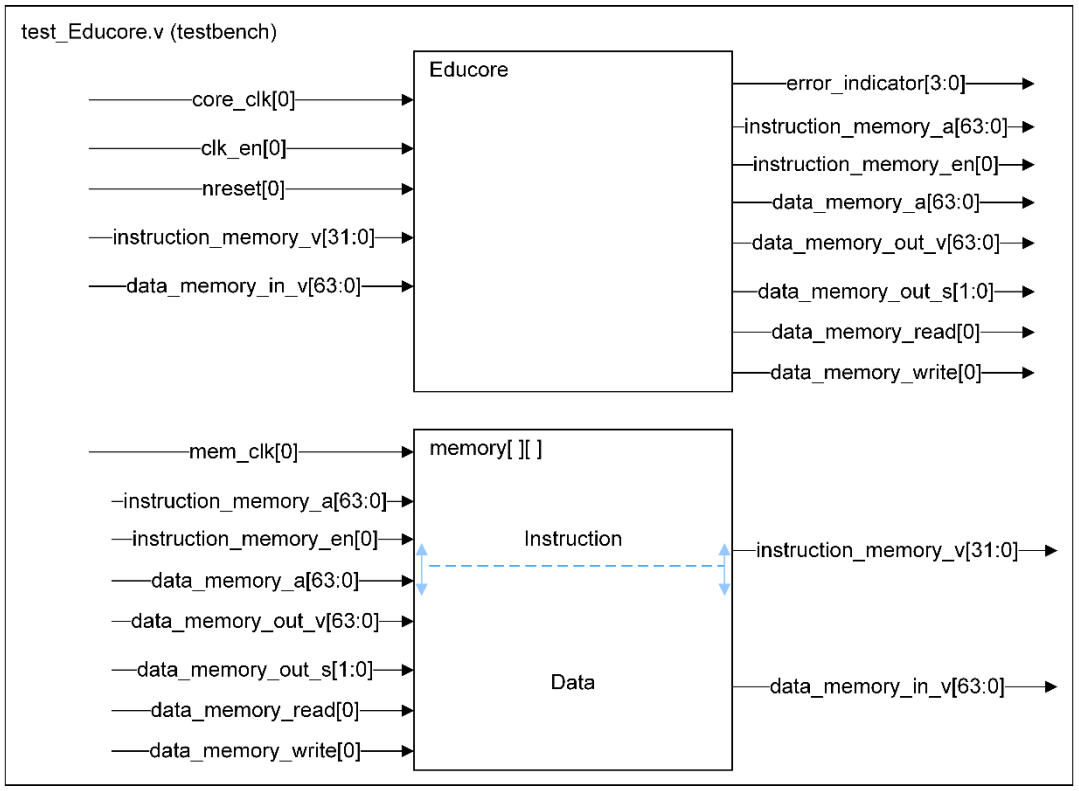


Figure 1: High-level view of the testbench and module **Educore**. *The testbench contains a unified memory containing both instruction and data. Note that the instructions are always at the top of the memory.*

We will learn more about these signals in Task: Verifying Assembly code on Arm Education Core.

Clock and ratios

The testbench defines 3 different clocks:

- **main_clk**: This is the main clock running at simulation tick.
- **core_clk**: This is the clock feeding into the module **Educore** (Arm Education Core).
- **mem_clk**: This is the clock feeding into memory in the testbench.

To simplify the memory implementation for Arm Education Core in this lab without requiring the need for asynchronous bridges between **Educore** and memory, we have set **core_clk** to never be faster than **mem_clk** to prevent data loss. If the **mem_clk** is too slow relative to **core_clk**, the memory may miss events coming from **Educore**. For now, **do not change**

the clock ratios in the testbench file unless instructed in the labs. Changing the clock ratios may produce unpredictable results.

Memory

Although Arm Education Core has separate busses for instruction and data accesses, the testbench uses a unified memory. This means both instruction and data are stored in a 64KByte 2D array (8 bit by 2^16) called **memory**. **The unified memory has no memory protection or hard boundary to separate instruction and data; therefore, your Assembly code must not cause an overlap between instructions and data (see Figure 2)—especially since the array is 8 bit wide and so a 32-bit instruction will easily take up 4 entries.**

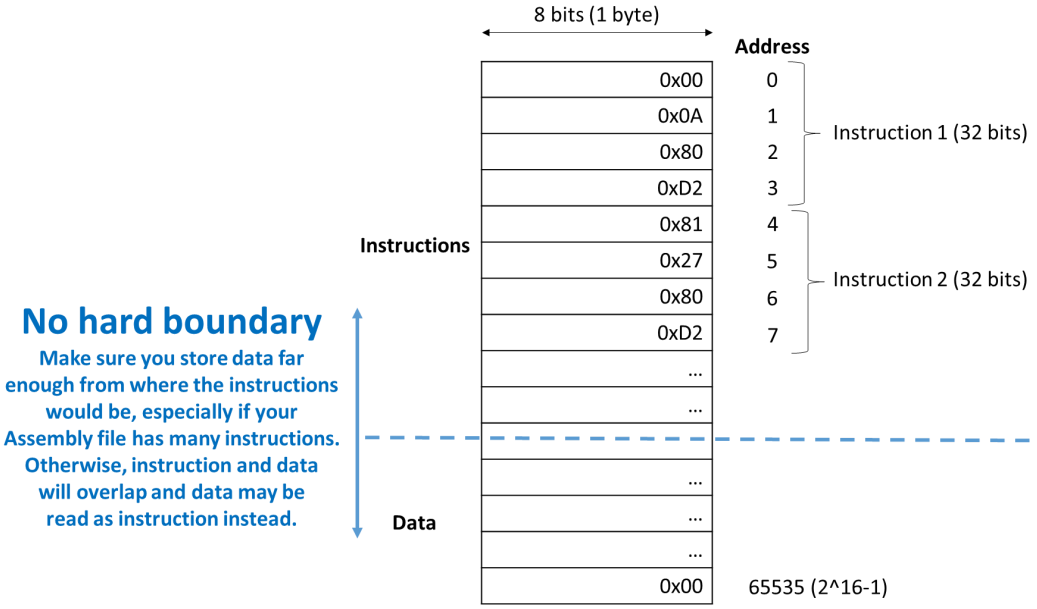


Figure 2: Unified memory in test_Educore.v file. The instructions are always at the top of the memory. The data can be placed at any location specified by the user. The light blue dashed line in memory indicates that there is no fixed boundary between instruction and data in the unified memory. Therefore, the user needs to ensure that any data stored (using the STURB instruction, for example) is placed in a far enough location from the instructions so that the instructions and data do not overlap.

Instruction read and data read/write are synchronized to the positive edge of **mem_clk**.

Note:

- The testbench loads the Assembly instructions .mem file using **$readmemh**.
- When **nreset=0**, instruction value in the testbench is assigned a NOP instruction to prevent Xs from being read into the single-cycle Arm Education Core.

Error indicator

The testbench also waits for an error indicator from the Arm Education Core:

- If **error_indicator** = 0, the simulation will not finish.
- If **error_indicator** = 1, there is an undefined error (e.g., an instruction that Arm Education Core does not support)
- **If error_indicator**= 2, Arm Education Core detected a **YIELD** instruction. You can place a **YIELD** instruction as the end of your Assembly source file to indicate that all instructions have been processed. **The testbench will halt simulation automatically when a YIELD instruction is detected.**

5.4.1 Task: Using Arm Education Core's testbench for simulation

To run an assembly code on Arm Education Core, you will first need to indicate to the testbench to finish simulation once all your instructions have been processed:

1. Modify **test_STRCPY.s** to add the **YIELD** instruction at the end of your Assembly code.

```
// store values in memory
STURB W5, [X0]
STURB W6, [X0, #1]
STURB WZR, [X0, #2]
YIELD
```

2. Save the file. **Recompile the Assembly source file and convert the ELF file into encodings**, similar to what was done in the previous exercise:

```
aarch64-none-elf-gcc -nostdlib -nodefaultlibs -lgcc -gdwarf-4
-Wa,-march=armv8-a -Wl,-Ttext=0x0 -Wl,-N -o test_STRCPY.elf
test_STRCPY.s
aarch64-none-elf-objcopy -O verilog test_STRCPY.elf test_STRCPY.mem
```

5.5 Task: Verifying Assembly code on Arm Education Core

To run an Assembly code on Arm Education Core, you first need to compile and simulate the Arm Education Core using Icarus Verilog. Follow these instructions:

1. Copy and extract **Educore-SingleCycle.zip** into your Lab_1 folder.

2. Open a Windows terminal and change directory into the unzipped **Educore-SingleCycle** folder.

   ```
   cd Educore-SingleCycle
   ```

   Note: In **Educore-SingleCycle** folder, there are 3 folders:
   - **src** contains the Verilog source files
   - **head** contains the Verilog header file
   - **tests** contains the testbench for the Arm Education Core.

3. Compile the Arm Education Core using the following command:

   ```
   iverilog -Wall -Wno-timescale -Wno-implicit-dimensions -I head/ -t
   vvp -y src/ -s test_Educore src/* tests/* -o test_Educore.vvp
   ```

Command/switch	Description
iverilog	**iverilog [option switches] [sourcefile]**
-Wall	Enable all warnings
-Wno-timescale	Disable warnings for inconsistent use of timescale directive
-Wno-implicit-dimensions	Disable warnings for creation of implicit declarations
-I	Specify include directory for header file
-t	Specify target (vvp)
-y	Append directory to module search path
-s	Specify top-level module
-o	Specify output file name

4. Simulate Arm Education Core using the following command, noting that test_STRCPY.mem is in the directory hierarchy 1 level above:

```
vvp test_Educore.vvp -lx2 +TEST_CASE=../test_STRCPY.mem
```

Command/switch	Description
vvp	**vvp [option switches] inputfile [extendedargs]**
-lx2	Dumps out waveforms in lx2 (LXT2) format
+TEST_CASE=../test_STRCPY.mem	The + operator passes the arguments to the vvp file, which would be picked up by $value$plusargs in the testbench file.

5. When the simulation is complete, a **dump.lx2** file is created, and you should get the following message in the windows terminal:

```
[EDUCORE LOG]: Test case: test_STRCPY.mem
LXT2 info: dumpfile dump.lx2 opened for output.
[EDUCORE LOG]: Apollo has landed
```

Note: If you are getting the message `[EDUCORE ERR]: Houston, we got a problem` instead, ensure that you have inserted the **YIELD** instruction and recompiled and generated the correct .mem file.

6. Use GTKWave to check contents on the waveform by using the following command:

```
gtkwave dump.lx2
```

Note: If you want to run GTKWave in the background (similar to Unix command "&"), use the `start "" gtkwave dump.lx2` command in your Windows terminal.

7. Add the following signals to view the waveform:

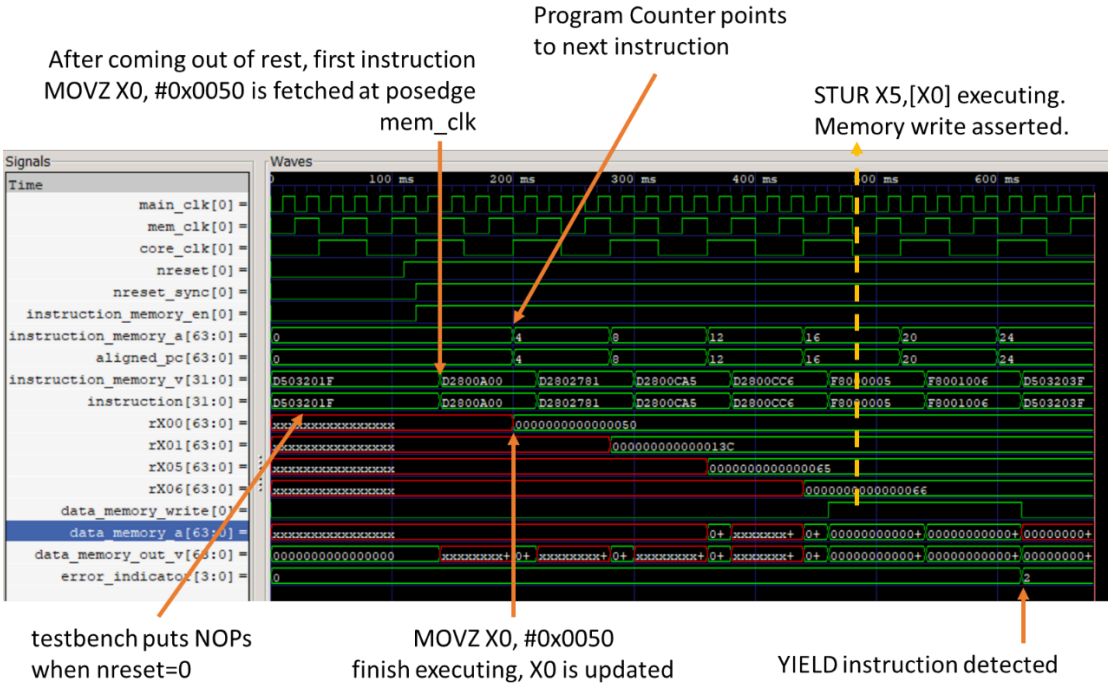
<module name>/<signal name>	Description
test_Educore/core_clk	Clock signal for Arm Education Core
test_Educore/mem_clk	Clock signal for memory in testbench
test_Educore/nreset	Active-LOW asynchronous reset signal in Arm Education Core
Educore/nreset_sync	Synchronizes nreset to next rising edge of core_clk

Educore/instruction_memory_en	Instruction memory read enable signal
Educore/instruction_memory_a	Instruction memory address
Educore/aligned_pc	64-bit Program Counter of Educore
Educore/instruction_memory_v	Instruction value
Educore/instruction	Instruction fetched
register_file/rX00, rX01, rX05, rX06	Registers X0, X1, X5, and X6
test_Educore/data_memory_write	Data memory write control. Set to 1 to write data.
test_Educore/data_memory_out_v	Data memory write value
test_Educore/data_memory_a	Data memory address
test_Educore/error_indicator	0—no error (no halt) 1—undefined error 2—YIELD instruction detected

8. In the GTKWave window, select **File > Write Save File As** and save as the name **debug.gtkw**. This will save the signal filters that you can reuse in other simulations, instead of dragging the signals manually again. Ensure that this file is in the same folder location as **dump.lxt2**. Then, click OK (we will be using this file in the next exercise).

9. Verify that the register contents are as expected, as shown in the snapshot below.

   Note: You can right-click on the signal and select Data Format to Decimal or Hexadecimal.



Observations:

- When nreset=0, the testbench puts NOPs into instruction_memory_v—this was a workaround to prevent Arm Education Core from reading in Xs briefly after coming out of reset.
- Arm Education Core's program counter (**aligned_pc**) points to the address of the instruction to be fetched. **aligned_pc** increments in a multiple of 4. This is because the instruction width is 4 bytes (32 bits).
- At the rising edge of **core_clk**, register X0 (rX00) has already been updated by value 0x50, corresponding to the first instruction (**MOVZ X0, #0x0050**). At the end of the second cycle, register X1 has been updated.
- When `STURB X5, [X0]` is executing, the **data_memory_write** signal is asserted:
  - **data_memory_out_v** = 0x65
  - **data_memory_out_a** = 0x50

5.6 Exercise: Creating a string copy operation

In this exercise, we will create a string copy operation.

1. Copy the following code for _strcpyloop (shown in blue below) into **test_strcpy.s** file so that the contents of the file are as shown below. Ensure that the **YIELD** instruction is the last in the file.

```
.global _start
.text
_start:
//place move instructions here
MOVZ   X0, #0x0050
MOVZ   X1, #0x013C
MOVZ   X5, #0x65
MOVZ   X6, #0x66

// store values in memory
STURB W5, [X0]
STURB W6, [X0, #1]
STURB WZR, [X0, #2]

//strcpy operation
_strcpyloop:
LDURB  W2, [X0]  // Load byte into X2 from memory pointed to by X0 (*src)
ADD X0, X0, #1 // Increment src pointer
STURB  W2, [X1]  // Store byte in X2 into memory pointed to by X2 (*dst)
ADD  X1, X1, #1     // Increment dst pointer
CMP   X2, #0         // Was the byte 0?
BNE   _strcpyloop    // If not, repeat the _strcpyloop
YIELD
```

The string copy operation first loads a byte from memory address pointed by the source pointer. It then increments the source pointer by 1 and then stores the loaded byte into the memory at an address pointed by the destination pointer. The destination pointer is then incremented by 1. This whole operation repeats and only ends after the byte loaded and stored has the value 0.

Note:

- Arm Education Core only has limited functionality and only supports a limited subset of the Armv8-A ISA. Arm Education Core does not currently support the `LDR <Xt>, [Xn]` instruction as this can be assembled in many ways of which the opcode is unsupported by Arm Education Core. However, Arm Education Core supports the **LDUR** (Store Register Unscaled) instruction, which can give the same outcome. There are also specific **LDUR** instructions depending on the data size to store, such as **LDURB** and **LDURH**.
- The difference between **LDR** and **LDUR** is that the **LDUR** is the *Load (unscaled) Register* instruction that allows accessing 32/64-bit values that are not aligned to the size of the operand. This is unlike the **LDR** operation, which requires the immediate offset to be in multiples of 4 or 8. However, **LDR** and **LDUR** pre-index instructions in this case gives the same result; therefore, we will be using **LDUR** for Arm Education Core instead.

a. What does the **LDURB** instruction do? How is **LDURB** different from **LDUR**? (**Hint: See the A64 Base Instructions in** [Additional references])

2. Save the file and recompile using **aarch64-none-elf-gcc** and the switches shown in the previous exercise.

```
cd Lab_1

aarch64-none-elf-gcc -nostdlib -nodefaultlibs -lgcc -gdwarf-4
-Wa,-march=armv8-a -Wl,-Ttext=0x0 -Wl,-N -o test_STRCPY.elf
test_STRCPY.s

aarch64-none-elf-objcopy -O verilog test_STRCPY.elf test_STRCPY.mem
```

3. Provide a snapshot of its register content (including Register X2) waveforms below to show that it is working.

```
cd Educore-SingleCycle

vvp test_Educore.vvp -lx2 +TEST_CASE=../test_STRCPY.mem

gtkwave dump.lx2 debug.gtkw
```

4. Reduce the number of instructions in **_strcpyloop** so that it does not require the use of ADD to increment the pointers. Recompile the Assembly code, generate .mem file, and re-simulate on Arm Education Core to verify correct behavior. (**Hint: Arm Education Core supports the post-index increment variant for LDRB and STRB instructions as the load/store post-index instruction type has a distinct encoding, and therefore do not get assembled in various ways. See A64 Base Instructions in Additional references.**)

6 Summary

In this lab, we have run an Assembly code on the single-cycle Arm Education Core. We did a simple string copy operation using Armv8-A instructions. However, how does Arm Education Core recognize these instructions in order to process them? Each instruction is encoded, as we have seen in the generated .mem file. In the next lab, we will interpret these encodings and observe how immediate values are encoded in the instruction itself.

7 Additional references

A64 Base Instructions

- https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions?lang=en

The A64 instruction set

- https://developer.arm.com/documentation/den0024/a/The-A64-instruction-set

A64 General-Purpose Registers

- https://developer.arm.com/architectures/learn-the-architecture/armv8-a-instruction-set-architecture/registers-in-aarch64-general-purpose-registers

Zero register

- https://developer.arm.com/documentation/den0024/a/ARMv8-Registers/AArch64-special-registers/Zero-register
- https://developer.arm.com/documentation/den0024/a/An-Introduction-to-the-ARMv8-Instruction-Sets/The-ARMv8-instruction-sets/Registers
- https://developer.arm.com/documentation/den0024/a/ARMv8-Registers

MOVZ instruction

- https://developer.arm.com/documentation/100076/0100/a64-instruction-set-reference/a64-general-instructions/movz

Icarus Verilog User Guide, commands, and arguments

- https://iverilog.fandom.com/wiki/User_Guide
- https://iverilog.fandom.com/wiki/Iverilog_Flags

VVP commands and arguments

- https://iverilog.fandom.com/wiki/Vvp_Flags

Simulation

- https://iverilog.fandom.com/wiki/Simulation

GTKWave manual

- http://gtkwave.sourceforge.net/gtkwave.pdf

arm-none-eabi-as command

- https://manned.org/arm-none-eabi-as/a7ae4940

Arm Cortex-A Series Programmer's Guide for Armv8-A

- https://developer.arm.com/docs/den0024/a/preface

GNU manual

- https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html

Arm Education Core

- Introduction to Arm Education Core document (provided in the Getting Started folder)