Exercises are based on material from Besim Mustafa, Edge Hill University.

Programming Model: Lab Exercises

Exercise 1 - Simple arithmetic operation and stack

To get acquainted with the simulator, we are going to program a simple arithmetic addition: calculating 5+8. Besides this simple calculating, we are also going to manipulate the stack.

A) First, create two instructions that **moves** the values 5 and 8 to register R00 and R01, and execute these instructions(executing can be done by double-clicking the instruction).

Instructions:	MOV #5, R0 MOV #8, R1

B) Next, make an instruction which **adds** the contents of R00 and R01. Execute it, and observe what happens. Where is the result placed?

Instruction:	ADD R00,R01
Observation:	The result is placed in the second register of the instruction.

C) Store the result on the **Program Stack** by making an instruction that **pushes** the result on top of the stack. Also push value -2 on the stack. Observe what happens to the **SP**.

Instruction:	PSH R01 PSH #-2
Observation:	The Stack Pointer is increased, it keeps pointing to the top of the stack.

D) Finally, write an instruction which **Pops** the top value of the Program Stack into register R02. Observe the SP again. What happens when you execute the command multiple times?

Instruction:	POP R02
Observation:	Stack pointer decreases. When stack is empty, a stack underflow occurs.

Exercise 2 - Conditions and jumps

We are going to write instructions that use conditions, and use it to make jumps in the program. You should continue with the program made in Exercise 1.

A) First, make an instruction that compares the contents of registers R00 and R01. Observe the value in the **SR**. Explain what the flags mean.

Instruction:	CMP R00,R01
Observation:	None of the flags are checked: no **Ov**erflow, no **Z**ero and no **N**egative value.

B) Next, manually add values to R04 and R05. Observe what happens when R04 and R05 are equal, and what happens when R04 is larger than R05.

Instruction:	CMP R04, R05
Observation:	When R04 and R05 are equal, the Zero flag is set: When the values are subtracted, the result is zero. When R04 is larger than R05, the Negative flag is set: When the values are subtracted, the result is negative

C) Next, make an instruction to **jump** unconditionally to the start of the program. Observe the values in the **PAdd** and **Ladd** columns. What do these addresses mean?

Instruction:	JMP 0
Observation:	PAdd is the physical address as viewed from the perspective of the hardware. LAdd is the logical address as viewed from the perspective of the program

D) Observe the LAdd value of the first and second instruction of the program. What is the difference between them, and what does this value indicate?

Difference:	It is equal to the instruction length in bytes

E) Modify the instruction of C) to only jump when R04 and R05 are equal, and test it by executing it.

Instruction:	(CMP R04, R05) JEQ 0
Observation:	The jump uses the instruction before itself in its condition

With jumps and compare instruction, a simple loop could be made. The to-be-designed loop will increment register R06 5 times, and then push it to the stack. To make jumping easier, a label could be added to the start of the loop, instead of using the address of the instruction.

Placing a label could be done by using the "NEW LABEL" button in the window where new instructions are selected. To use the label in a jump command, select the label in the dropdown menu.

```
Code:   MOV #0, R06
        LOOP1:
        ADD #1, R06
        CMP #5, R06
        JNE $LOOP1
        PSH R06
        HLT
```

Using a jump to a label makes the jump very flexible, since the label can be placed anywhere in the program. Note the usage of "$" when a label is used with a jump instruction.

F) Analyze the code above, and insert it into the simulator. Observe how it works.

Observation:	The code clears the contents of register R06. It then increments R01 with 1, and compares it with 5. When not equal, a jump is made to the start. Otherwise the push instruction is executed, in this case 5 is pushed on the stack.

Exercise 3 - Direct and indirect addressing

Instead of directly writing to a register or another place in memory, also the value of a register could be used to access the memory. This is called indirect accessing.

A) Find the instruction in the **Appendix** which could be used to store one byte of data in a memory location. Make and execute an instruction that stores number 65 (all numbers are decimal) on address location 20.

Instruction:	STB #65, 20

B) Make and execute an instruction that stores number 22 into register R01, and a second instruction that uses the value of R01 as address to stores number. Note the use of "**@**" as prefix for R01.

Instruction:	MOV #22, R01 STB #51, @R01

C) Observe the contents of data memory(especially locations 20 and 22). Refer to image 9.

Observation:	The memory contains 41 and 33, which are the hex values of the inserted numbers

Exercise 4 - Subroutines

Let's create a simple subroutine. Enter the following code in the simulator. A new label **PRINT** needs to be created at the start of the subroutine: this label represents the starting address of the subroutine. Adding the label is done in the same way as described earlier.

Make sure to select the "**Direct Mem**" option with the **OUT** command

Code:
```
PRINT:
OUT 24, 0
RET
```

The subroutine outputs text present at memory location 24 and returns (see **RET** instruction in the appendix). To use the subroutine, some text needs to be present at address location 24.

- Click on the **SHOW PROGRAM DATA MEMORY**… button (see Image 6). In the displayed window highlight the line with 0024 under the **LAdd** column.
- Under **Initialise Data**, click on the **String** button.
- Enter some text in the textbox labelled **Value**, e.g. *My name is …*
- Click the **UPDATE** button

The subroutine needs to be called to be useful. This is done by using a combination of the instructions **MSF** and **CAL** (refer to the Appendix). The MSF (Mark Stack Frame) is need to reserve a place for the return address on the program stack. The CAL instruction needs to specify the starting address of the called subroutine.

The subroutine can be combined with the loop from exercise 2:

Code:
```
LABEL0:
MOV #0, R06
LOOP1:
ADD #1, R06
MSF
CAL $PRINT
CMP #5, R06
JNE $LOOP1
HLT
PRINT:
OUT 24, 0
RET
```

A) Follow the instructions above, and insert the code into the simulator. Run the code and observe what happens.

Observations:	The code prints the text from the memory 5 times.

B) The PRINT subroutine uses a fixed address. Change the subroutine (and the code in the LOOP1) so indirect memory addressing is used: the value present in register **R07** is used. (Make sure when testing to set R07 and to place a new string in the corresponding data memory location)

Instructions:

```
LABEL0:
MOV #24, R07
MOV #0, R06
LOOP1:
ADD #1, R06
MSF
CAL $PRINT
CMP #5, R06
JNE $LOOP1
HLT
PRINT:
OUT @R07, 0
RET
```

C) (optional) Finally, change the instructions so the loop itself also becomes a subroutine.

Instructions:

```
LABEL0:
MOV #24, R07
MSF
CAL PRINT5
HLT
PRINT5:
MOV #0, R06
LOOP1:
ADD #1, R06
MSF
CAL $PRINT
CMP #5, R06
JNE $LOOP1
RET
PRINT:
OUT @R07, 0
RET
```

Exercise 5 – Additional exercises (optional)

To review your knowledge about the basics of the simulator, you could make the following exercises.

Enter the instructions you create in order to answer the questions in the blank boxes. Refer to Appendix at the end of this document to find the details on the desired instructions. You are expected to execute the instructions you created on the simulator in order to verify your answers. Your tutor(s) will be available to help you if you require assistance.

A) Loops using jump and compare instructions:

1. Write a conditional statement such that if R02 is **greater than** (>) R01 then R03 is set to 8. (Use R01 as the first operand and R02 as the second operand)

```
CMP R01, R02
JGT $MOV8
HLT
MOV8
MOV #8, R03
```

2. Write a conditional statement such that if R02 is **less than or equal to** (<=) R01 set to -5. (Use R01 as the first operand and R02 as the second operand)

```
CMP R01, R02
JGT $MOVMIN5
HLT
MOVMIN5
MOV #-5, R03
```

3. Write a conditional statement such that if R01 = 0 then R03 is set to 5 **else** R03 is set to R01 plus 1.

```
CMP #0, R01
JEQ $L0
MOV R01, R03
ADD #1, R03
HLT
L0
MOV #5, R03
HLT
```

4. Write a loop that repeats **5 times** where R02 is incremented by 2 every time the loop repeats.

```
MOV #0, R01
L0
ADD #2, R02
ADD #1, R01
CMP #5, R01
JLT $L0
HLT
```

5. Write a loop that repeats **while** R04 is > 0. Set the initial value of R04 to 8.

```
MOV #8, R04
L0
CMP #0, R04
JGT $L0
HLT
```

6. Write a loop that repeats **until** R05 is > R09. Set the initial values of R05 to 0 and R09 to 12.

```
MOV #0, R05
MOV #12, R09
L0
CMP R05, R09
JGT $L0
HLT
```

7. Write a routine that **pushes** numbers 8 and 2 on top of stack. It then **pops** the two numbers one by one from stack, **adds** them and **pushes** the result back to top of stack.

```
Program Tut3:    0014 POP R01
0000 MSF         0017 POP R02
0001 CAL $L0     0020 ADD R01, R02
0005 HLT         0025 POP R03
0006 L0          0028 PSH R02
0006 PSH #8      0031 PSH R03
0010 PSH #2      0034 RET
```

8. Challenge #1: Place 15 numbers from 1 to 15 on top of stack using the **push** instruction in a loop. Then in a second loop use the **pop** instruction to pop two numbers from top of stack, **add** them and **push** the result back to top of stack. The second loop repeats this until there is only one number left on top of stack which should be the final result.

```
Program Chal1:
0000 MOV #1, R01
0006 L0
0006 PSH R01
0009 ADD #1, R01
0015 CMP #15, R01
0021 JLE $L0
0025 L1
0025 POP R02
0028 POP R03
0031 ADD R02, R03
0036 PSH R03
0039 DEC R01
0042 CMP #2, R01
0048 JGT $L1
0052 HLT
```

B) Instructions for writing to and reading from memory (RAM):

The following instructions access the program's data memory. You can display this memory so that you can observe the results by referring to Image 1 and the related text in section D.

9. Locate the instruction that **stores** a byte in memory and use it to store number 65 in memory address location 20 (this uses **memory direct** addressing method).

```
0000 STB #65, 20
```

10. Move number 51 into register R04. Use the store instruction to **store** the contents of R04 in memory location 21 (this uses **register direct** addressing method).

```
STB R04, 21
```

11. Move number 22 into register R04. Use this information to indirectly **store** number 58 in memory (hint: you will need to use the '@' prefix for this – see the list of instructions in appendix) - (this uses **register indirect** addressing method).

```
MOV #22, R04
STB #58, @R04
```

12. Locate the instruction that **loads** a byte from memory into a register. Use this to load the number in memory address 22 into register R10.

```
LDB 22, R10
```

C) Putting it together:

13. Challenge #2: write a loop in which 10 numbers from 48 to 57 are stored as single bytes in memory starting from memory address 24. You should use **register indirect** addressing method of storing the numbers in memory (see exercise 11 above).

```
Program Chal2:
0000 MOV #24, R02
0006 MOV #48, R01
0012 L0
0012 STB R01, @R02
0017 INC R01
0020 INC R02
0023 CMP #57, R01
0029 JLE $L0
0033 HLT
```

14. Challenge #3: write a loop in which the numbers stored in memory by the program in (13) above are copied to a different part of the memory starting from address location 80.

```
Program Chal3:
0033 MOV #80, R03
0039 MOV #24, R02
0045 L1
0045 LDBI @R02, R01
0050 STBI R01, @R03
0055 CMP #34, R02
0061 JLT $L1
0065 HLT
```