Programming Model 4

Omar Mian, mqg069

Objectives of this tutorial

At the end of this lab you should be able to:

- Enter CPU instructions using the CPU simulator
- Describe the effect of compare instruction on CPU status flags Z and N
- Construct a loop and explain jump instructions that use Z and N flags
- Use direct addressing to access memory location(s)
- Use indirect direct addressing to access memory location(s)
- Construct a subroutine and call it
- Pass parameter(s) to the subroutine

Special instructions

- Save your work at regular intervals.
- The CPU simulator is work in progress so it may crash from time to time. If this happens then restart it and load your latest code, hence the importance of the above statement!
- Ask if you need help with any aspect of this tutorial.
- The simulator is there to help you understand the theory covered in lectures so make the most of it.

Exercises

Learning objectives: To enter CPU instructions and describe effect of compare instruction on CPU status flags Z and N.

In theory	In practice

The CPU instruction sets are often grouped together into categories containing instructions with related functions.

This exercise familiarises you with the way the instructions are entered using the simulator.

The most encountered instructions involve data movements, e.g. the **MOV** instruction.

Another common instruction is the comparison instruction, e.g. **CMP**. This instruction sets or resets the status flags **Z** and **N** as a record of the result of the comparison operation.

1. Enter the following code and run it:

```
MOV #1, R01
MOV #2, R02
ADD R01, R02
```

2. Add the following code and note the states of the status flags Z and N after each compare instruction is executed:

```
CMP #3, R02        Z  = 1     N  =0
CMP #1, R02        Z  =0      N  =0
CMP #4, R02        Z  =0      N  =1
```

Explain your observations of the states of the status flags after the compare instructions above:

CMP #3, R02 → Z = 1, N = 0

- This compares the value 3 (immediate) with R02 = 3
- Since they are equal, the result is zero, so the zero flag is set to 1
- The result is not negative, so the negative flag remains 0

CMP #1, R02 → Z = 0, N = 0

- Compares 1 with R02 = 3 → 3 - 1 = 2
- The result is non-zero so Z = 0
- The result is positive so N = 0

CMP #4, R02 → Z = 0, N = 1

- Compares 4 with R02 = 3 → 3 - 4 = -1
- The result is non-zero so Z = 0
- The result is negative so N = 1

**** Now save the above code! ****

Learning objectives: To construct a loop and explain jump instructions that use Z and N flags

In theory	In practice
The computer programs often contain loops where the same sequence of code is repeatedly executed until or while certain condition is met. Loops (or iterative statements) are the most useful features of programming languages. At the instruction level, loops use conditional jump instructions to jump back to the start of the loop or to jump out of the loop. This exercise is created to demonstrate the use of the jump instruction which often uses the result of a compare instruction. Such jump instructions use the **Z** and the **N** status flags to jump or not jump.	3. Add the following code and run it (ask your tutor how to enter a label): **MOV #0, R01** **Label1** **ADD #1, R01** **CMP #3, R01** **JLT $Label1** Summarize what the above code is doing in plain English: - It sets R01 = 0. - Then it adds 1 to R01. - After each addition, it compares R01 with 3. - If R01 is less than 3 (JLT = jump if less than), jump back to Label1 and keep adding 1. - When R01 reaches 3, the condition fails (no longer less than 3), and the program moves past the loop. 4. Modify the above code such that R01 is incremented by 1 until it reaches the value of 4. Copy the code below: MOV #0, R01 Label1 ADD #1, R01 CMP #4, R01 JLT $Label1 **** Now save the above code! ****

Learning objectives: To use direct addressing to access memory location(s)

In theory	In practice

Although instructions moving data to or from registers are the fastest instructions, it is still necessary to move data in or out of the main memory (RAM) which is a much slower process. Examples of instructions used to store into or get data out of memory are explored here. The method used here uses the <u>direct addressing</u> method, i.e. the memory address is directly specified in the instruction itself.	5. Add the following code and run: **STB #h41, 16** **LDB 16, R03** **ADD #1, R03** **STB R03, 17** Make a note of what you see in the program's data area: At memory address 16, the value is stored, which is the ASCII code for A Then, the program loads that value into R03, and adds 1 to it which makes it h42, which is the ASCII code for B, and stores it at memory address 17. Result: Memory[16] = 'A' Memory[17] = 'B' What is the significance of **h** in **h41** above? The h means that the number is in hexadecimal Modify the above code to store the next two characters in the alphabet. Write down the modified code below: STB #h43, 18 ; Store 'C' at address 18 STB #h44, 19 ; Store 'D' at address 19 **** Now save the above code! ****

Learning objective: To use indirect addressing to access memory location(s)

In theory	**In practice**

There are circumstances which make direct addressing unsuitable and inflexible method to use.

In these cases indirect addressing is a more suitable and flexible method to use. In indirect addressing, the address of the memory location is not directly included in the instruction but is stored in a register which is the used in the instruction.

This exercise introduces an indirect method of accessing memory. This is called register indirect addressing. There is also the memory indirect addressing but this is left as an exercise for you.

6. Add the following code and run:

```
Label2
MOV #16, R03
MOV #h41, R04
Label3
STB R04, @R03
ADD #1, R03
ADD #1, R04
CMP #h4F, R04
JNE $Label3
```

Make a note of what you see in the program's data area:

Starting at memory address 16, the program stores a sequence of characters.
It starts by storing 'A' (ASCII h41) at address 16.
Then 'B' (h42) at address 17, 'C' (h43) at 18, etc...
It adds 1 to both the memory address and the character each time.
This continues until the character reaches h4F, which is 'O'.
Summary:
Memory[16] = 'A'
Memory[17] = 'B'
Memory[18] = 'C'
...
Memory[29] = 'O'
(Addresses 16 through 29 get filled with letters A–O.)
Explain the significance of **@** in **@R03** above:

It means indirect addressing. Instead of directly writing to a hard coded address, whatever address is available in R03 is dynamically used.

**** Now save the above code! ****

Learning objective: To construct a subroutine and call it

In theory	**In practice**

Another very useful feature of programming languages is subroutines. These contain sequences of instructions which can be executed many times. If a subroutine was not available the same sequence of instructions would have been repeated many times increasing the code size. At instruction level, a subroutine is called by an instruction such as **CAL**. This effectively is a jump instruction but it also saves the address of the next instruction. This is later used by a subroutine return instruction, e.g. **RET** to return to the previous sequence of instructions.	7. Add the following code but do **NOT** run it yet: **MSF** **CAL $Label2** Now convert the code in (6) above into a subroutine by inserting a **RET** as the last instruction in the subroutine. Make a note of the contents of the **PROGRAM STACK** after the instruction **MSF** is executed (<u>see tutor what to do to facilitate this observation</u>): Make a note of the contents of the **PROGRAM STACK** after the instruction **CAL** is executed: The program stack is empty after MSF, which clears any old return addresses or data from the stack. What is the significance of the additional information? The return address is important for program flow control. Without saving the return address during CAL, the CPU would get lost and not know where to resume after the subroutine finishes. Storing this address makes sure that the program returns where it left off. **** Now save the above code! ****

Learning objective: To pass parameter(s) to the subroutine

In theory	**In practice**

Useful as they are the subroutines are not very flexible without the use of the subroutine parameters. These are values passed to the subroutine to be consumed inside the subroutine.

The common method of passing parameters to the subroutines is using the program stack. The parameters are pushed on the stack before the call instruction. These are then popped off the stack inside the subroutine and consumed.

This exercise is created to demonstrate this mechanism.

8. Let's make the above subroutine a little more flexible. Suppose we wish to change the number of characters stored when calling the subroutine. Modify the calling code in (7) as below:

```
MSF
PSH #h60
CAL $Label2
```

Now modify the subroutine code in (6) as below and run the above calling code:

```
Label2
MOV #16, R03
MOV #h41, R04
POP R05          ←
Label3
STB R04, @R03
ADD #1, R03
ADD #1, R04
CMP R05, R04     ←
JNE $Label3
RET
```

Add a second parameter to change the starting address of the data as a challenge and write the code in the box above!

```
Label2
POP R03 ; Pop starting memory address into R03
POP R05 ; Pop ending character into R05
MOV #h41, R04 ;Start with A
Label3
STB R04, @R03 ; Store the character at the address in R03
ADD #1, R03 ; Increment memory address
ADD #1, R04 ; Increment character
CMP R05, R04 ; Compare with end character
JNE $Label3 ; Loop if not done
RET
```

**** Now save the above code! ****