Department of Electrical and Computer Systems Engineering
ECE2071 Systems Programming

Assignment 3: MIPS Assembly

Introduction

Assembly language forms an important part of this unit as it provides a gateway to understanding computer architecture and some of the lower level functions of computers. In this unit we will use MIPS as an example of an assembly language and also explore its relationship with the C programming language.

This assignment has been split into 5 sections;

- Part 1: Debugging Quiz
- Part 2: Introductory MIPS tasks
- Part 3: More Complex MIPS Problems
- Part 4: Interacting with Peripheral Devices
- Part 5: The Stack

This assignment contains a variety of styles of task, including coding, multiple choice and short answer questions. Ensure that you read carefully and follow the submission instructions on the submission page for this assignment on Moodle. The skeleton code for this assignment can be downloaded from this link.

Learning Objectives

By the end of this assignment you will be able to:

- Understand the functioning of instructions in the MIPS Instruction Set Architecture
- Load data from memory and store data to memory using MIPS instructions
- Effectively debug MIPS instructions
- Implement bit masking approaches using MIPS instructions to manipulate numbers
- Construct programs using MIPS instructions that interact with peripheral devices

Environment, Assessment, and Submission

You will submit your work via a Moodle dropbox. You should submit the `src` folder of your assignment (the only folder provided in the skeleton download) as a zipped folder, using the instructions below. Note a penalty will apply for not following these instructions or the naming conventions.

Your work will be assessed using test cases (possibly with different numbers to detect hard coding).

Outside of this process, note that a human may 'spot check' or review your work and award limited partial credit, or apply deductions on the basis of it not meeting specifications or coding standards.

Your assignment must be completed individually, and following the generative AI policy listed on the assignment dropbox. If you use resources to assist you (from inside or outside the unit, or from a discussion with a colleague) you must acknowledge and cite them in your work.

Submission

The skeleton code for this assignment is essentially a zipped file that contains a single folder named `src`, within which startup codes are provided for the tasks in this assignment. You should work on the tasks in this assignment using the startup codes provided on the MARS MIPS simulator. Once you have completed all the coding tasks in this assignment, copy the updated `src` folder containing your MIPS assembly codes into a directory `assignments/assignment_3` within your home directory in VS Code.

You will submit a zipped folder containing the `src` directory of your assignment. In order to zip your `src` directory, run the following command from the `assignment_3/src` directory. Note that `<student_id>` in the command should be replaced with your unique Student ID.

```
~/assignments/assignment_3/src$ zip -r <student_id>_assignment_3.zip *
```

For example, if your student ID is 12345678, you will run the following (the command you run is bolded):

```
~/assignments/assignment_3/src$ zip -r 12345678_assignment_3.zip
*
  adding: mem_acc.asm (deflated 58%)
  adding: mips1.asm (deflated 49%)
  adding: mult_add.asm (deflated 39%)
  adding: prime_detector.asm (deflated 58%)
  adding: count_alternating_bits.asm (deflated 57%)
  adding: reverse_array.asm (deflated 59%)
  adding: stack_usage.asm (deflated 50%)
  adding: check_structure.py (deflated 52%)
```

This will create a zip file named studentID_assignment_3.zip in your `src` directory. You should then upload this file to the Assignment 3 submission box on Moodle.

Penalties will apply for not submitting a zipped file in the correct format.

Note: If you have additional files in your `src` directory, this is okay.

You can check whether your zipped file is in the correct format by running the `check_structure.py` python script (in the skeleton code) from the `src` directory by specifying the path to the zipped file as command line argument:

You can check whether your zipped file is in the correct format by running the `check_structure.py` python script (in the skeleton code) from the `src` directory by specifying the path to the zipped file as command line argument:

```
~/assignments/assignment_3/src$ python3 check_structure.py 12345678_assignment_3.zip
Structure correct!
```

If you are missing files, or have them in the wrong place, this script will print which ones are in incorrect locations, and you can correct this and re-zip your submission.

The submission of this zipped file accounts for 2.5/5 of the total marks available for Assignment 3. The remaining 2.5/5 marks are allocated to the associated quiz on Moodle. Note that you should also complete the quiz before the due date.

Marking

In order to mark your assignment it will be run using an automatic tester, which will assemble and run your MIPS assembly programs under a variety of test inputs. You will **not** be provided with a sample test script for this assignment. You are expected to verify the correctness of each task by inspection and manual testing.

Ensure you obey the specifications of the task regarding which registers to use

The marks allocated to the coding tasks in this assignment (2.5/5) are as follows:

mult_add (Task 2.1) - 0.25 mark
prime_detector (Task 3.1) - 0.75 mark
count_alternating_bits (Task 3.2) - 0.75 mark
reverse_array (Task 5.1) - 0.5 mark
stack_usage (Task 5.2) - 0.25 mark

Task 1: MIPS Basics and Debugging

In this exercise, you will work to debug a series of short MIPS assembly programs.

1.1 Simple Addition Program

In this section, you will debug a simple addition program, in a file named `mips1.asm` within the `src` folder in the skeleton code. Fix the errors in this program and step through the program line by line to examine its execution. **The errors identified in this program will need to be recorded as your answer to Question 1 in the Assignment 3 - Quiz on Moodle**.

```
# Written for ECE2071 S1 2024
# Copyright Monash University 2024

# Fix the errors in this program and examine its execution

.text                          # Start generating instructions
.globl start                   # This label should be globally known

start  li $t0, 0x1             # Load the value 1
       li $t1, 0x2             # Load the value 2
       addi $t2, $t0, $t1      # Add the values, store in $t2
       sub $t3, $t1, t0        # Subtract value 1 from value 2
                               # and store in $t3
infinite:
      jump infinite            # an infinite loop so that the
                               # computer will not execute code
                               # past the end of the program
```

Question: Are any of the MIPS instructions in the code above pseudo instructions?
The answer to this question will need to be recorded as your answer to **Question 2 in the Assignment 3 - Quiz on Moodle**.

1.2 Accessing Memory

We will now try to debug the following MIPS assembly program that attempts to load from and store to memory using different MIPS instructions. This code below has been provided to you in a file named `mem_acc.asm` within the `src` folder in the skeleton code. You will need to fix a couple of syntax errors that are present before the code can run. **The errors identified in this program will need to be recorded as your answer to Question 3 in the Assignment 3 - Quiz on Moodle**.

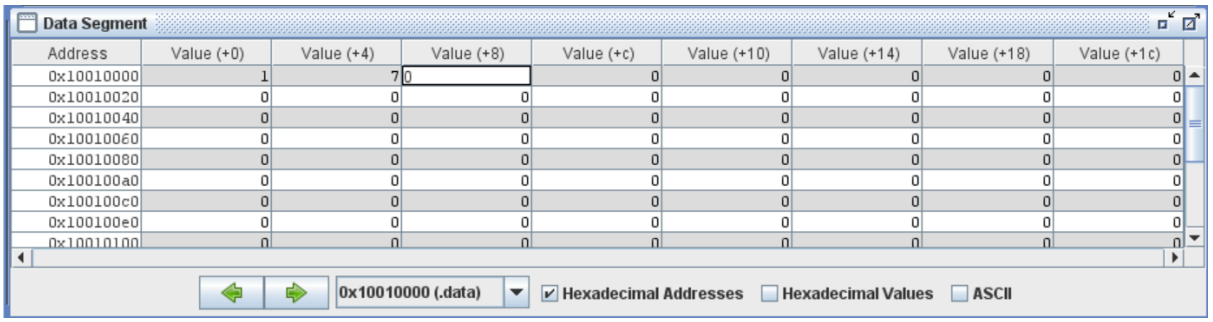
The program initially loads three different memory locations into three different registers. It then loads data from one memory location into a particular register, and then stores back the same data from the register to a new memory location utilizing three different types of load-store instructions on the MIPS processor. The loading and storing process is then repeated indefinitely using an infinite loop.

```
# Written for ECE2071 S1 2024
# Copyright Monash University 2024

.text                      # Start generating instructions
.globl start               # This label should be globally known

start:
      lui t1, 0x1001       # Load upper half of memory address,
                                 # lower half is filled with zeros

      li $s0, 0x10010004   # Load second memory address as 32-bit value
                           # into $s0 using li pseudo-instruction
```

```
        lui $s1, 0x10010008 # Load third memory address into $s1

repeat
        lw $t0, 0x0($t1)    # Load 32-bit word from the first memory address INTO
register $8/$t0
        sw $t0, 0x0($s0)    # Write 32-bit word to the second memory address FROM
register $8/$t0

        lhu $t2, 0x0($t1)   # Load 16-bit halfword (interpreted as unsigned) INTO
register $t2
        sh $t2, 0x0($s1)    # Write 16-bit halfword to third memory address FROM
register $t2

        lbu $t3, 0x3($t1)   # Load 8-bit byte (unsigned) from an offset of 3 from
first memory address
        sb $t3, 0x4($s1)    # Store 8-bit byte to the next word address after
third memory address
        j Repeat

.data                       # Items below this line will be stored in the
                            # .data section of memory
```

Once the errors in the program have been fixed and the program has been assembled successfully, execute it step by step and investigate what happens to the registers, program counter (PC register), and values in the data segment on the simulator.

You can initialize the value stored at the memory locations by first assembling the program, and then double clicking the desired memory locations in the “Data Segment” section of the simulator, as shown in the image below.



Task 2: Introductory MIPS Problems

2.1 Multiplication by Successive Addition

In the file `mult_add.asm` within the `src` folder in the skeleton code, complete the MIPS assembly program to multiply two numbers (loaded into `$t0` and `$t1` registers), and store the product in the `$s0` register. You should only add instructions to the skeleton code provided under the `main` label in the code. In addition, your program should end in an infinite loop.

Note: You must use the registers specified above or you will not receive marks.

You can use the C code below as a guide on what your MIPS assembly program should do:

```c
// start section (already completed for you):
int t0 = 2;
int t1 = 3;

// main section:
int s0 = 0;
while (t1 > 0) {
    s0 = s0 + t0;
    t1 = t1 - 1;
}
// end section:
// infinite loop
while (1) { }
```

Skeleton code:

```
# Written for ECE2071 S1 2024
# Copyright Monash University 2024
# Multiplication via successive addition

.text                   # Start generating instructions
.globl main             # Makes the label main globally known
.text                   # Instructions in the program:
start:
# setting $t0 = 2 and $t1 = 3:
addi    $t0, $0, 2
addi    $t1, $0, 3
```

```
# main program:
main:
# WRITE YOUR CODE HERE
```

Do **not** modify code that is above the line "# WRITE YOUR CODE HERE", as this may interfere with the automarker, and result in a loss of marks for this task.

2.2 Basic Bitwise Operations

As we have seen in the Week 7 lab activity (of the ECE2071 project) where bitwise manipulations were performed to break a 16-bit ADC output of the STM32 microcontroller into two bytes for data transmission, it can be useful to be able to simply transform a number in terms of its individual binary bits.

In this section, we will investigate how to deploy some of the bitwise operations that we have covered in the Week 7 pre-workshop videos to perform certain operations on a number. Answers for this section should be recorded as your answer to **Questions 4 - 7 in the Assignment 3 - Quiz on Moodle**.

Suppose we have a single binary digit (bit), which can have a value of either 1 or 0. Consider how the value of a bit changes depending on its initial value and upon being subjected to the following operations:

a. AND with 0
b. OR with 0
c. XOR with 0
d. AND with 1
e. OR with 1
f. XOR with 1

Based on your understanding of the Week 7 content in the unit, fill out the table below with the modified value of a bit depending on its initial value and the bitwise operation to which the bit is subjected to, and identify any patterns that exist.

Operation	**Initial value of Bit = 0**	**Initial value of Bit = 1**
AND with 0		
OR with 0		
XOR with 0		
AND with 1		
OR with 1		
XOR with 1		

If the registers on the MIPS processor were initialized as follows (note that prefix of b' refers to binary or base 2 representation):

$t1 = b'1110
$t2 = 15
$t3 = 12
$t4 = 9
$t5 = b'1010

Identify the value that will be loaded into register $v0 after the execution of each of the following instructions:

(a) and $v0, $t1, $zero
(b) ori $v0, $t5, 15
(c) xor $v0, $t3, $t3
(d) and $v0, $t4, $t3

Which single true assembly instruction (not pseudo or P-type instruction) on the MIPS instruction set architecture could you use to obtain the one's complement of the value in register $t1 and then store the result in register $v0.

$t1 = b'0101

Task 3: More Complex MIPS problems

3.1 Prime Number Detector

In Task 2.1, we tried to perform multiplication of integers via repeated MIPS add instructions. However, this becomes very slow for performing multiplications on large numbers. As such, the MIPS instruction set architecture supports instructions for both multiplication and division, namely `mult` and `div`. These are special instructions, as they use special hardware to perform the operations, and the results of these instructions are always stored in the `HI` and `LO` registers. To access these `HI` and `LO` registers, two more special instructions are used: `mfhi`, to move from the `HI` register into a general-purpose register, and `mflo`, to move from the `LO` register into a general-purpose register. They can be used as follows:

```
mult    $t0, $t1
mflo    $t2          # $t2 now contains $t0 * $t1

div     $t3, $t4
mflo    $t5          # $t5 now contains $t3 / $t4
mfhi    $t6          # $t6 now contains $t3 % $t4
```

In the file named `prime_detector.asm` within the `src` folder in the skeleton code, you have been provided with startup code to load a value into register `$a0`, call the `is_prime` function, and then end in an infinite loop. Your task is to complete the `is_prime` function.

The input argument is provided to the function in register `$a0`. The result should be returned in register `$v0`. The 'result' of the function is 1 if the number in `$a0` is prime, and 0 if the number in `$a0` is not prime.

Note:

- **You must use the registers specified or you will not receive marks**
- For your reference, a number is **prime** if its only two factors are 1 and the number itself
- 0 and 1 are **not prime** numbers
- You can assume that we are dealing with positive numbers

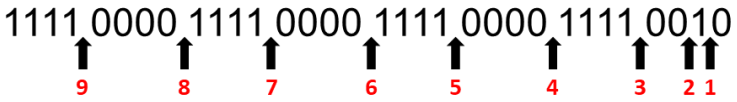
3.2 More Bitwise Operations

In a file named `count_alternating_bits.asm` within the src folder in the skeleton code, your task is to write a MIPS function, named `count_alternating_bits` that counts the number of instances where bits alternate in the sequence loaded into register $a0. The function should then return the count in register $v0.

For example:

If $a0 = 0xF0F0F0F2 = 1111 0000 1111 0000 1111 0000 1111 0010 in binary,

$v0 should be set as 9 upon exit from the function since there are 9 instances where bits alternate within the sequence as shown in the image below.



Note: You must use the registers specified or you will not receive marks

Task 4: Interacting with Peripheral Devices

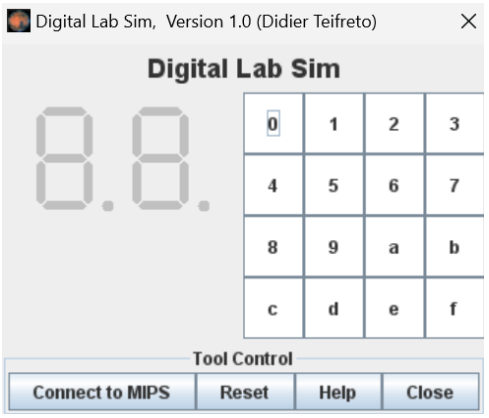
Introduction to Peripheral Devices

A peripheral device is any hardware device that can connect to a computer and provide input or output information. Things such as a keyboard or a mouse are considered peripheral devices. There are essentially two main ways of performing input/output operations between a CPU and peripheral devices: memory mapped and separate I/O. We will mainly focus on memory mapped I/O since this is the style that the MARS MIPS simulator uses. At the same time, note that separate I/O is also used by other architectures such as x86. Memory mapped I/O is defined by the use of the same address space for both memory and peripheral devices. Therefore, the same MIPS instructions that we use to access memory can be used to access I/O devices. Separate I/O requires a different set of instructions to access peripheral registers as they are not considered a part of normal memory.

4.1 Interacting with Peripheral Devices in MARS

MARS MIPS simulator comes packaged with a number of add-on peripheral device simulators. One of them is the “Digital Lab Sim”, which has simulated seven segment displays and keyboard buttons.
For this task, we will be using one of the seven segment displays and the number pad/keyboard.

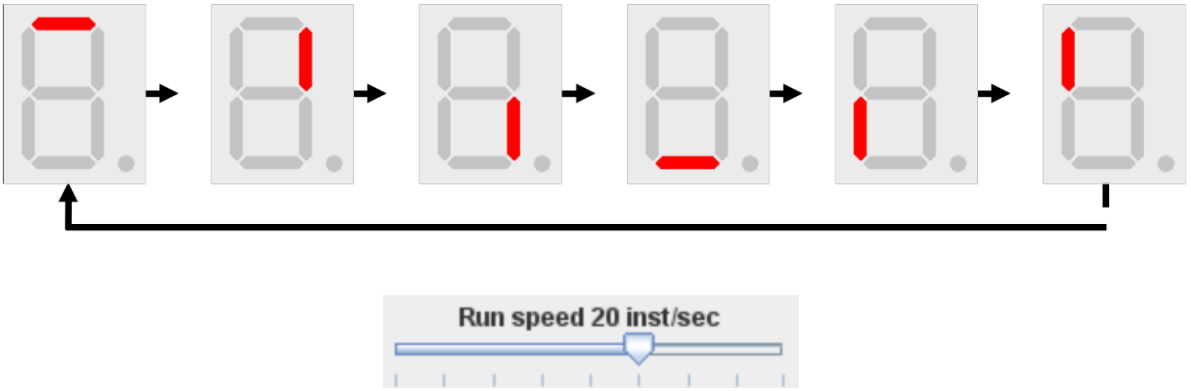
The Digital Lab Sim tool can be opened via the “Tools” menu (Tools -> Digital Lab Sim). The seven segment displays are memory-mapped output devices. This means that they are each associated with an address in memory, and writing to that address will control their state. The display that we will be using is mapped to address `0xFFFF0010`. Each bit in the byte of memory that is associated with the display will control one segment of the seven segment display. For example, writing the 8-bit binary value `00000001` to address `0xFFFF0010` in memory will turn ON the top segment of the display. Further, writing `00000010` will turn ON the top right segment, `00000100` will turn ON the bottom right segment, and so on.
The Digital Lab Sim is shown in the image below.



The code below is intended to write to each outer segment of the 7-seg display in a rotating pattern as shown on the next page.

```
# Revised for ECE2071 S1 2024
# Copyright Monash University 2024
# Rotating 7-Seg Example

.text                   # Start generating instructions
.globl start             # Makes the label main globally known

.text                   # Instructions in the program:
start:  li $t0, 0xFFFF0010    # Load memory address of seven segment display
        li $t1, 0x1       # Initialise value to write to display

repeat: sb $t1, 0($t0)          # Write the value 0x1 to seven segment display
        sll $t1, $t1, 1
```

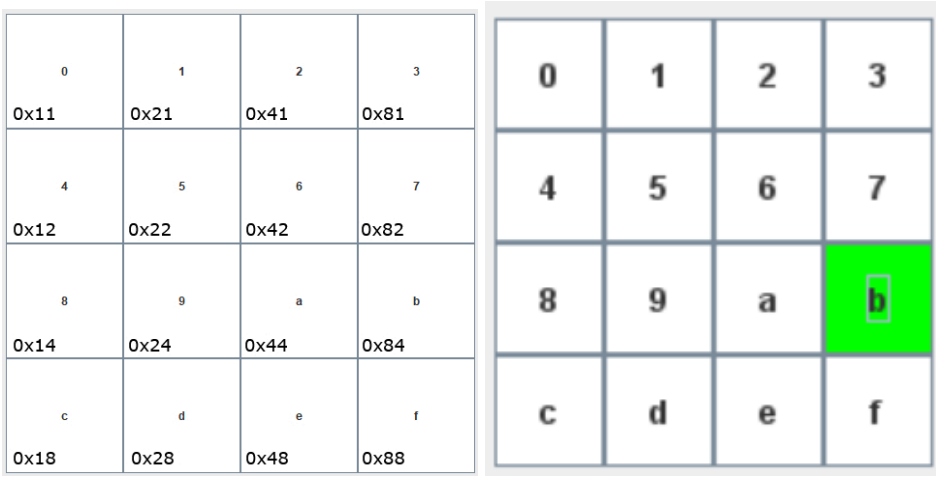
```
        blt $t1, 0x40, repeat
        j start     # Branch to repeat label
```



Before running the code above, make sure to open the Digital Lab Sim and click the "Connect to MIPS" button at left. You can slow down the speed at which instructions get executed by using the "Run speed" slider at the toolbar of the simulator.

To run the code, ensure it has been assembled without error. Then open the Digital Lab Sim tool and select 'Connect to MIPS'. You may now run the program however it is **recommended to slow down the execution** (i.e. to 10-20 inst/sec) to better observe what the code is doing.

Using the memory-mapped number pad is a little more complicated. There are two key byte-sized memory locations in this case, which we'll refer to as the *scan* byte and the *values* byte. A button on the number pad can be pressed by clicking on it. Only one button on the number pad can be pressed at any point of time. In order to release a pressed button, it needs to be clicked again.



The lower 4 bits of the *scan* byte refers to a row (as 2^{row}) on the number pad. We will use the *scan byte* to check whether any button in a particular row has been pressed by the writing the value of 2^{row} to the memory location 0xFFFF0012. For example, we can check whether any button located within row 2 of the number pad (such as the *button b* pressed in the image on the previous page) is pressed by writing a *scan byte* of 0x4 (2^2) to the memory location 0xFFFF0012.

The lower 4 bits and upper 4 bits of the *values* byte for each button indicates the row (as 2^{row}) and column (as 2^{column}), respectively of a button of the number pad. We will use the *values byte* to identify whether a button has been pressed within the scanned row on the number pad by reading the byte stored at memory location 0xFFFF0014. For example, if the *button b* at row 2 and column 3 of the number pad is pressed (as shown in the image on the previous page), the *values* byte will be 0x84 where 0x4 (lower 4 bits) = 2^2 and 0x8 (upper 4 bits) = 2^3.

To read from the number pad, first write the scan byte that corresponds to the row of the number pad being checked. For example, write 0x4 or 0000 0100 (in binary) to memory location 0xFFFF0012 in order to check for row 2 or the third row of the number pad. You can then read from the *values* byte at memory location 0xFFFF0014 to identify which particular button on row 2 has been pressed.

Now that you're familiar with Digital Lab Sim, you have all the knowledge required to complete the following task.

4.2 Designing a Program that Interacts with Peripheral Devices

In **Question 8 of Assignment 3 - Quiz on Moodle**, a program has been provided that is supposed to take a user input from the first row of the number pad (0….3), convert the input into an appropriate form and then display it on the right seven-segment display in Digital Lab Sim. The number should keep being displayed after it is clicked and the display should only change when a new number has been clicked. You should complete this program and verify its functionality on the MARS simulator.

Note: Run your solution to this problem at a low speed (10-20 inst/sec) on Digital Lab Sim in order prevent the MARS simulator from stalling

Task 5: The Stack

5.1 Working with Memory

In this task, we will be working with arrays in MIPS. You will be writing a function that reverses an array.

Arrays are stored as a continuous section of memory, where each element is stored sequentially in the reserved section of memory. In higher level languages, such as C, we typically use an extra variable to store the length of the array. However, in a MIPS processor, we only have a limited number of registers, and therefore we will use a **different** approach to

store the size of the array: we will store the size of the array as the first element of the array, and the remaining elements will be stored after it. Thus, to store an array of length N, we must use N+1 locations in memory: the first location to store the length of the array, and the next N locations to store the value of the array elements.

Consider the following example representing the first few memory locations of the data segment in the simulator. The data segment below contains an array of length 4, starting at position 0x10010000

Address	Value(+0)	Value(+4)	Value(+8)	Value(+c)	Value(+10)	Value(+14)	…
0x10010000	4	654	324	42	78	…	…
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

The grey cells above indicate the section of memory that is used for the array.

In a file named `reverse_array.asm` within the `src` folder of the skeleton code, implement the function named `reverse_array`, that does the following:

- Takes in the address of an array in memory, which will be formatted in the conventions described above (the first element is the length of the array, and the subsequent elements are the contents of the array)
- Reverses the order of elements in the array (keeping the length of the array stored at the initial memory location)
- Returns to the calling function

Each array element will be stored as a **word** (4 bytes). You have been provided with skeleton code that initialises an array at memory location 0x10010000, however you cannot always assume that the array will be located at this location.

After reversing the array provided as example in the diagram above, the resulting array should look like the following:

Address	Value(+0)	Value(+4)	Value(+8)	Value(+c)	Value(+10)	Value(+14)
0x10010000	4	78	42	324	654	…
⋮	⋮	⋮	⋮	⋮	⋮	⋮

5.2 Stack

In a file named `stack_usage.asm`, using the $sp pointer and conventions shown in the Week 7 pre-workshop video 3, store the integers 1 to 10 on the stack. For full marks you must ensure that the stack pointer moves in the **conventional correct direction**, and the allocated size of the stack is the **minimum** required for the number of 32-bit words stored.

You may only use registers $zero, $t0, $t1, $t2 and $sp in your solution.