Queen Mary
University of London

ECS404U: Computer Systems & Networks

2020/21 – Semester 1

Prof. Edmund Robinson, Dr. Arman Khouzani

Lab Week 08: *Introduction to MIPS Assembly*

November 09/11, 2020

Deadline for submitting your proof of work: ***Next week's Thursday, at 10:00 AM UK time***

Student Name:

Yujeong JUNG

Student ID:

200699145

Brief Feedback to student (commendations, areas for improvement):

Summary and Learning Objectives

This is the first of three labs on MIPS architecture and Assembly language. The learning objectives of today's lab are the following:

- Familiarity with the 32-bit MIPS architecture (general purpose registers, instruction formats);
- How to write simple MIPS Assembly programmes in a (code-friendly) text editor like **`Atom`** and executing them in the **`QtSpim`** MIPS-simulator;
- Learning MIPS Assembly instructions for:
  - loading constant values into registers;
  - basic integer arithmetic (adding, subtracting);
  - basic logic operations (AND, OR, XOR);
- Understanding the difference between immediate (I) and register (R) instruction formats;
- Practise how to combine multiple instructions to achieve more complex operations;

1 Preparation

In this and the following two labs, we will be using a programme called **`QtSpim`**. It is a free (under BSD License) and open-source simulator of a simple 32-bit processor called the "MIPS R3000".[1] It allows, for educational purposes, to see what happens when a MIPS-based CPU executes its machine-level instructions. In particular, it lets us see the content of all registers and main memory while we step through the instructions.
Recall that the programming language that is the closest to the machine code is called **assembly** (or sometimes **assembler**) language. A utility programme called the *assembler* converts the assembly instructions (which are just "mnemonics" for their binary representation) to executable binaries (i.e., machine codes) through just a simple look-up. Of course, each processor with a different ISA would have its own distinct assembly languages (there are as many assembly languages as there are ISAs created). Here, we will be dealing with MIPS assembly.
Typically, the assembly codes are generated by a *compiler*, which is a programme that translates (in a not-so-easy way!) from a high level code (like C, C++, Fortran, Lisp, Pascal, VB, *etc.*) to the mnemonics instructions of an ISA, i.e., its assembly language. However, we will be directly writing assembly codes ourselves (to learn!).
For writing any code, including assembly, we can use any text editor. However, some text editors have some nice features for writing codes, e.g., automatic highlighting based on a syntax, auto-completing snippets (by clicking the tab key or showing a list of options), syntax checks, automatic indentation, *etc.* Great examples of such text editors with code-development features that are free and open-source are **`Atom`** (MIT License), VScode (MIT License), **`Eclipse`** (under Eclipse Public License).

1.1 Installing `QtSpim`

To install **`QtSpim`** on your machine, first download the latest version of the pre-compiled installable file relevant to your Operating System from its *sourceforge* directory at
https://sourceforge.net/projects/spimsimulator/files/.
At the time of writing, these are the latest versions:

[1]Fun fact: SPIM is MIPS spelled backwards. Also **`Qt`** is the name of a free cross-platform framework for creating graphical user interfaces, and yes, the pun with "cute" is intentional!

Windows: `QtSpim_9.1.21_Windows.msi`.

macOS: `QtSpim_9.1.21_mac.pkg`.

Linux (Deb): `qtspim_9.1.22_linux64.deb`.

For **windows**, just double-click on the downloaded installer file and follow its steps. Installation may take up a minute while it seems the window has frozen, but be patient, and it should finish. After which, you should find **`QtSpim`** under your programme files, and/or from the desktop depending on your choice during the installation.
For **macOS**, you may also run the downloaded installer file. However, if you receive an error that the package is from an *unidentified developer*, then the most secure way to install it is by using `Homebrew`, which is a package manager programme for macOS. To check if you already have it installed, open a terminal and run `brew`. If you get the *command not found* error, then you can install it by copy/pasting the command provided in https://brew.sh/ (which reads like `/bin/bash -c "$(curl -fsSL https://.../install.sh)"`) into your terminal (and hit enter!). When it is done, run the following additional two commands in the terminal:

```
brew install caskroom/cask/brew-cask
brew cask install qtspim
```

Note that you may be prompted for your password.
For **Linux**, first see if double-clicking on your downloaded file starts your package installer. If it doesn't then you can open a terminal and run the following command:

```
sudo dpkg --install ~/Downloads/qtspim_9.1.21.1_linux64.deb
```

of course changing the path to the wherever you downloaded the `deb` to! After installation is finished you can run **`QtSpim`** either from command line or from your programme menu (like any other programme!)

> If you have issues with your installation after you have tried hard, ask one of the demonstrators to help you or post your error message on our discussion forum.

1.2 Configuring `QtSpim`

If your installation finished successfully, you should now be able to open **`QtSpim`**. We are going to make some adjustments in **`QtSpim`** to make the interface less crowded and easier to interpret:

- From the menu bar, under "`Text Segment`", make sure "`Kernel Text`" is **unchecked** (and the rest are checked).
- From the menu bar, under "`Simulator`", choose "`Settings`", then from the "`MIPS`" tab, **uncheck** the "`Load Exception Handler`".
- From the menu bar, under "`Simulator`", choose "`Run Parameters`". Make sure the value of the field "`Address or label to start running program`" is **`0x00400000`** (and not for instance, `0x00000000`).

1.3 Installing and Configuring `Atom`

Among the choices for our code editor, we provided the instructions for installing and configuring **`Atom`**, although any of them (or even a simple text editor like notepad!) would do.
Download and run the installer of **`Atom`** that is relevant to your OS from its website: https://atom.io/.
After that, we need to also install language support for MIPS assembly (for syntax highlighting and auto-completing snippets) in it: Open **`Atom`**, and go to `Edit` → `Preferences` → `Install`, and search for **`MIPS`**, and install the package named **`language-mips`**.

2 Review: Basics of MIPS architecture (Please read carefully!)

What is MIPS? MIPS is a simple architecture for a CPU (so it is not itself a chipset, but rather the 'blueprint' to design one). It is simple only in the sense that it has implemented very few instructions in its hardware. So few instructions that the list of them can fit in a single page! However, it is not simple in the sense of its functionality: It is a general purpose computer. This means that it can run whatever programme you can think of: an internet browser, a word processor, a web server, even a modern computer game (as a matter of fact, it was the processor used in Nintendo 64 and Sony PlayStation 2). Nowadays, it is still used in embedded systems such as home routers.[2] Note that although MIPS comes in a 64-bit flavour too, we are going to focus on its most prevalent 32-bit architecture.

RISC vs CISC Building a general computer based on very few simple instructions, so that each instruction takes very few clock cycles, is called **Reduced Instruction Set Computer (RISC)**. MIPS is an example of RISC design. Another notable example is ARM, which is the architecture of the microprocessor in almost all smartphones today. This is in contrast to **Complex Instruction Set Computer (CISC)**, where each instruction actually takes several hardware-level operations to perform. Notable examples of CISC is the AMD/Intel's x86 family, and the IBM's System/360 and z/Architecture.

MIPS Registers Recall that a register is a temporary storage unit built into the CPU itself. The hardware of the CPU uses registers to hold the values that it needs to perform an operation on (e.g. addition), and to put the result of the operation back in them. We saw inside of a single bit of a register before (recall: SRAM). Recall that it was built with cmos transistors, the same building blocks of the rest of the CPU.
MIPS has 32 integer registers[3], indexed from 0 to 31.
In a 32-bit MIPS, each (integer) register is 32 bits (4 Bytes) long (and this is why we call the architecture 32-bit as opposed to 64-bit, not the fact that incidentally, it has 32 registers). We call a set of bits that are processed at the same time a "**word**". So in a 32-bit MIPS, a **word** is 4 bytes long, and a **half-word** is 2 Bytes long.[4]
All MIPS registers are similar in their hardware, except for register \$0, whose bits are hardwired to zero (it always holds 32 bits of all zeros). It may sound bizarre, but we will shortly see why it is useful. Although the rest of the 31 registers are all available to us, the ISA has a convention on which registers are expected to use for what purposes. For ease of programming, each register is given a name. The list of the 32 registers along with their name and a short description of their usage convention is provided in Table 1. For this lab, we will only work with *temporary* registers \$t0 to \$t7 and *save* registers \$s0 – \$s7, as well as the *zero* register \$zero.

MIPS instruction formats As with the data, instructions are also represented in (pause for dramatic effect!), binaries. No surprise here, as we have no other choice: binary is the language of the hardware (read: transistors). Transistors are where the rubber meets the road, and they only react to high and low electric potentials. In a 32-bit MIPS processor, the instructions are 32 bits long. In fact, there is a register (other than the 32 registers above) whose job is to hold the value of the current instruction being executed, referred to as (guess what?!) the *Instruction Register (IR)*. These 32 bits should carry unambiguously all the information necessary for the CPU's hardware to know what operation it should perform, and on what operands (if applicable), and what do to with the result (if applicable).

[2]Fun fact: The pioneer of MIPS was David Patterson, the first author of our textbook.
[3]In computer architecture lingo, an array of registers in a CPU is sometimes referred to as the *register file*. Note that of course this has nothing to do with "files" as we know them!
[4]Your computer is likely a 64-bit architecture: so each "word" on your computer is 64 bits longs, i.e., the smallest unit of data that the CPU of your computer processes at a given time is 8 Bytes long.

Register Number	Register Name	purpose (usage)
\$0	\$zero	Always holds 32 bits of zeros
\$1	\$at	Reserved to be used by the assembler
\$2 – \$3	\$v0, \$v1	Return values from subroutines
\$4 – \$7	\$a0 – \$a3	Arguments to subroutines
\$8 – \$15	\$t0 – \$t7	Temporary registers
\$16 – \$23	\$s0 – \$s7	Save registers
\$24 – \$25	\$t8 – \$t9	More temporary registers
\$26 – \$27	\$k0 – \$k1	Reserved for Kernel (Operating System)
\$28	\$gp	Global Area Pointer
\$29	\$sp	Stack Pointer
\$30	\$fp	Frame Pointer
\$31	\$ra	Return Address

Table 1: MIPS integer registers.

The MIPS instruction follow 3 possible formats: "I", "R", "J" (standing for Immediate, Register, and Jump). Figure 1 shows the bit pattern of the "I" and "R" formats, which are the only two we need for this lab.

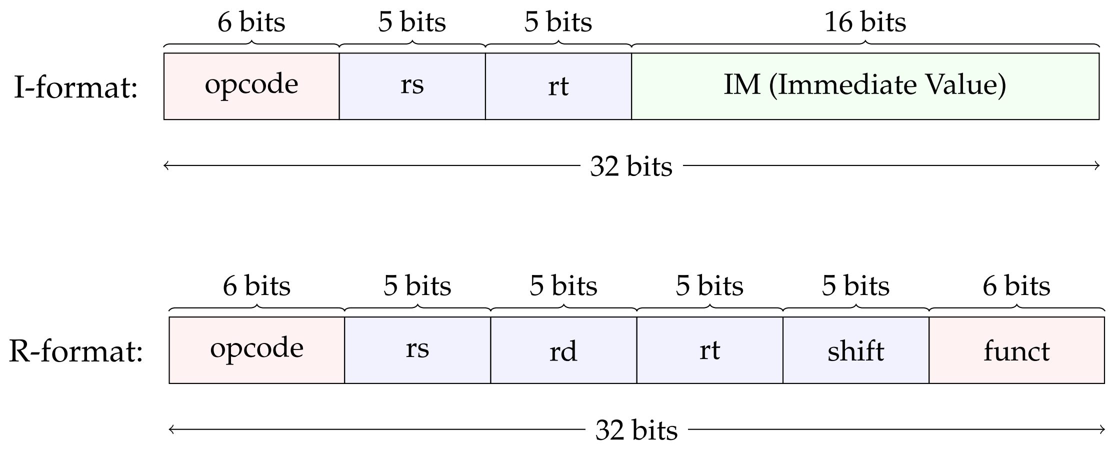


Figure 1: The MIPS instruction formats of "I" (above) and "R" (below).

Recall the following from the lecture:

- The "I" instructions operate on the content of a register and an *immediate* value directly given in the instruction; For example: 'add the immediate value of 4 to the integer in register **`rs`** and put the resulting integer in register **`rt`**'.
- The "R" instructions operate on the contents of two registers; For example: 'add the integer in register **`rs`** to the integer in register **`rt`** and put the result in register **`rd`**'.
- The **`opcode`** for "I" instructions, and the combination of **`opcode`**, **`shift amount`**, and **`funct`** for "R" instructions specify to the control unit which operation to perform (which inputs to connect to what part of ALU, and connect the output of the ALU to where).
- Most (but not all) "R" (3-register) instructions have an "I" (2-register and an immediate value) alternative.

3 Integer Arithmetic

3.1 addiu

Since we (humans) cannot as easily work with binaries directly, each instruction also has a **mnemonics** representation, which we can use to write our programs with.[5] **addiu** is an example of them: it is an "I" type instruction that does the following: takes the *immediate* value (which is part of the instruction itself), adds it to the integer in register **rs** (*source* register), and puts the result in register **rt** (*target* register). For instance,

```
addiu   $t0, $s0, 5
```

does the following: $\$t0 \leftarrow \$s0 + 5$. In order for us to understand our instructions, it is a good idea, but not mandatory, to add such comments to our codes. For instance, write:

```
addiu   $t0, $s0, 5     # t0=s0+5
```

(a) Writing our first MIPS Assembly programme

1. On **Atom**, create a new file, and write the following lines of text in it, and save it under the name `week8_3_1_a.asm`:

```
addiu   $t0, $zero, 1

addiu $v0, $zero, 10 # to Exit the program
syscall # Exit
```

Codes/week8_3_1_a.asm

2. The last two lines of instructions is just to normally exit the program (otherwise, the microprocessor continues executing whatever next is on the RAM!)
   NOTE: all of your assembly codes must have those two lines at the end.

3. In your **QtSpim**, click on "`File`" `->` "`Reinitialize and Load File`", and open the file `week8_3_1_a.asm` that we just created. You should see the screen as depicted in Figure 2. Study its different parts carefully!

4. You can now execute the program in "single steps" (one MIPS instruction at a time) by pressing the button marked as (6) in the figure (or pressing **F10**.)

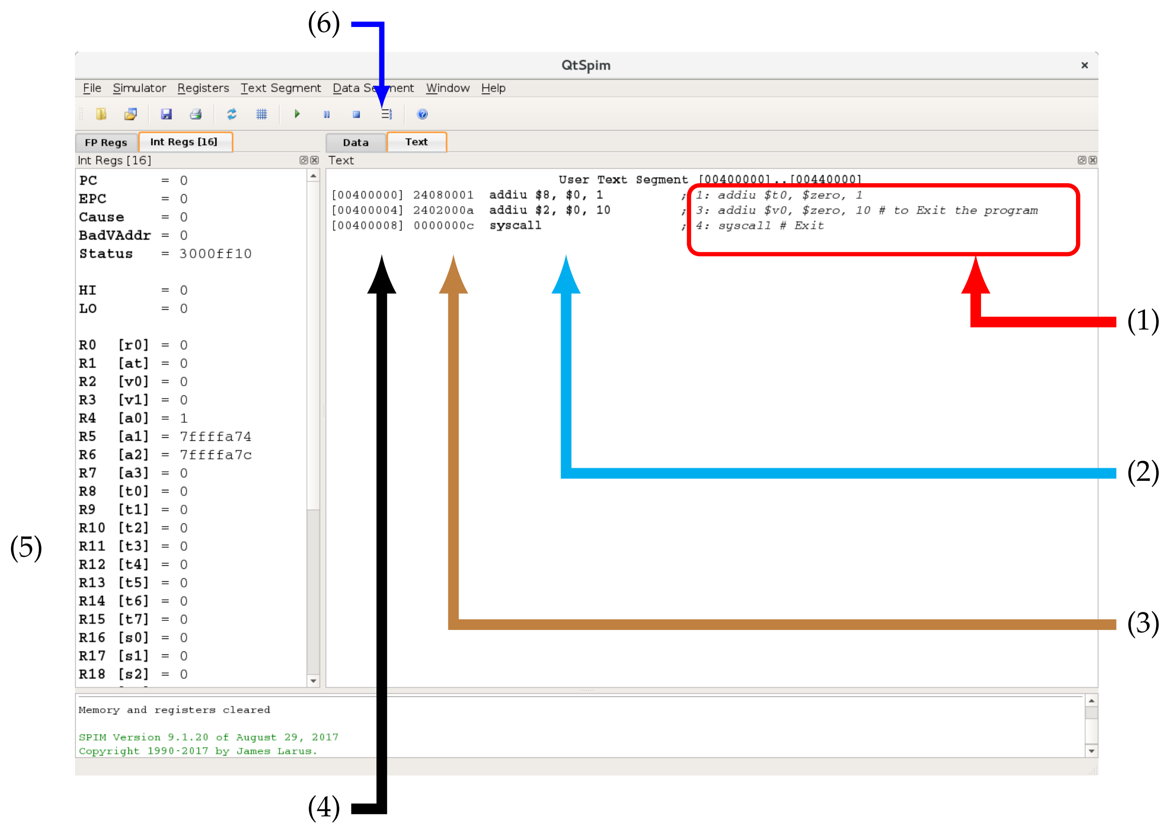
Note: for each new code (or if anything went wrong), you should always use "`File`" > "`Reinitialize and Load File`".

Now, answer the following questions.

Q.1: What does this program do (just by analysing the code)?

This program help to simulate the execution of MIPS assembly programs.
It checks a context and syntax, and adds needed instruction.
Also, it updates register and memory content as each instruction is executed.

[5] *mnemonics* literally are things that help us remember easier.



```
PC      = 0
EPC     = 0
Cause   = 0
BadVAddr = 0
Status  = 3000ff10

HI      = 0
LO      = 0

R0  [r0] = 0
R1  [at] = 0
R2  [v0] = 0
R3  [v1] = 0
R4  [a0] = 1
R5  [a1] = 7ffffa74
R6  [a2] = 7ffffa7c
R7  [a3] = 0
R8  [t0] = 0
R9  [t1] = 0
R10 [t2] = 0
R11 [t3] = 0
R12 [t4] = 0
R13 [t5] = 0
R14 [t6] = 0
R15 [t7] = 0
R16 [s0] = 0
R17 [s1] = 0
R18 [s2] = 0
```

```
                              User Text Segment [00400000]..[00440000]
[00400000] 24080001  addiu $8, $0, 1          ; 1: addiu $t0, $zero, 1
[00400004] 2402000a  addiu $2, $0, 10         ; 3: addiu $v0, $zero, 10 # to Exit the program
[00400008] 0000000c  syscall                  ; 4: syscall # Exit
```

```
Memory and registers cleared

SPIM Version 9.1.20 of August 29, 2017
Copyright 1990-2017 by James Larus.
```

Figure 2: The **QtSpim** interface, after you "reinitialize and load" the `week8_3_1_a.asm` file. Annotations:(1) our code (user text); (2) actual MIPS instructions corresponding to our code; (3) the 32 bits of the instruction; (4) the memory address (in RAM) where the instruction is fetched from; (5) the contents of the registers of the MIPS microprocessor; (6) single-step (one MIPS instruction at a time) execution of the instructions (equivalent to pressing **F10**).

Q.2: By looking up the instruction manual of MIPS (on QM+ or accessible from MIPS_Green_Sheet.pdf), construct the 32 bits representing the instruction in the first line of our programme, and break it into its constituents:

opcode :	001001
rs :	00000
rt :	01000
Immediate Value (in binary) :	1
the entire 32 bits :	00100100000010000000000000000001

Q.3: Why are there 5 bits to specify each register in a MIPS instruction?

The address for representing 32 registers only needs a total of 5 bits. Because the address is expressed in binary numbers, if you 2^5, get 32.

From the 404U w7 lecture slides;

Q.4: Convert the 32 bit of our instruction to hex.

```
0010 0100 0000 1000 0000 0000 0000 0001
2    4    0    8    0    0    0    1
=24080001
```

Q.5: Compare the results you obtained manually, to its value in **QtSpim** (get it from the column marked as (3) in Figure 2).

These two value are same.

Q.6: Now, run our program in steps (**F10**), and keep close track of the contents of the registers. Check that you get what you expect in register $t0. One of the special purpose registers internal to a processor keeps the address of the current instruction in the memory. This register is called **PC** (for Program Counter). What happens to **PC** when you step-execute your code? Explain! (specially, explain the value of the increments.)

$t0; there is no step.
In second one ($v0), there is one step and PC is 400004.
In last one (syscall), there is two step and PC is 400008.

Q.7: Recall that each register, including **$zero**, is 32 bits long. This should raise an eyebrow: the immediate value is only 16 bits long. How does a 16-bit value added to a 32-bit value (Note: the ALU of MIPS can only add two 32-bit inputs).

(From the w7 lecture slides and my note-taking)
Signed-extended;
the leftmost bit (the MSB) is replicated 16 times to the left.
When the immediate value is positive, the 16th bit is zero, and adding 16 more zeros behind it does not change its value.
When the immediate value is negative, the 16th bit is one, and adding 16 more ones

Q.8: What is the smallest and largest integer that the "immediate value" can have (Hint: investigate the format of the I-instructions)?

the smallest integer is -2^15
the largest integer is 2^15 -1

(b) Writing our second Assembly programme

Going back to **Atom**, write a new program ("File"->"New File"), similar to `week8_3_1_a.asm`, except that replace "1" with "-1". Save it under the name: `week8_3_1_b.asm`. Answer the following questions. (Hint: Recall negative integers are represented in 2's complement in a computer!)

Q.1: What does this programme do (just by analysing the code)?

This program help to simulate the execution of MIPS assembly programs.
It checks a context and syntax, and adds needed instruction.
Also, it updates register and memory content as each instruction is executed.

Q.2: **Without** looking up the instruction manual of MIPS, construct the 32 bits representing the instruction on the first line, convert it to hex and compare it to what **QtSpim** shows.

opcode :	001001
rs :	00000
rt :	01000
Immediate Value (in binary) :	-1
the entire 32 bits :	0010 0100 0000 1000 0000 0000 0000 1
its hex representation :	24080009
what **QtSpim** shows :	2408ffff

Q.3: As before, answer how does the microprocessor convert the 16-bit immediate value to 32 bits, so that it can be added to another 32 bit register?
Hint: To answer this, in **QtSpim**, under the "`Registers`" from the top menu, select "Binary" (as opposed to "Hex" or "Decimal") and inspect the content of register **t0**.
Does this transformation lead to a correct representation of -1 in 32 bits (2's complement)?

When Hex, t0 is 0.
When Binary, t0 is 0.
No, it doesn't carry a correct representation of -1 in 32 bits.

3.2 addu

This is "R" format companion of **addui**. Since it is an "R" instruction, it needs 3 registers to be specified. In particular:

```
addu    $t0 , $s0 , $s1    # t0=s0+s1
```

adds the integer value in register **s0** with the integer value in register **s1** and puts the result in register **t0**.

1. In **Atom**, create a new file and write the following code in it, and save it under the name of `week8_3_2.asm`:

```
addiu   $s0 , $zero , 1
addiu   $s1 , $zero , −1
addiu   $s2 , $zero , 2
addiu   $s3 , $zero , −2

addu    $t0 , $s0 , $s2
addu    $t1 , $s1 , $s3
addu    $t2 , $s1 , $s2
addu    $t3 , $s0 , $s3
addu    $t4 , $s2 , $s2
addu    $t5 , $s3 , $s3

addiu $v0 , $zero , 10 # to Exit the program
syscall # Exit
```

Codes/week8_3_2.asm

2. As usual, in **QtSpim** from "File" -> "Reinitialize and Load File", load `week8_3_2.asm`, but do not run it yet.

   Q.1: **Before** running the programme, determine what the final value of the following registers will be? (represented as their decimal value, except for **PC**, where you should predict its final Hex value)

t0 :	3	**t1** :	fffffffd
t2 :	1	**t3** :	ffffffff
t4 :	4	**t5** :	fffffffc
PC :	40002c		

   Execute the instructions step-by-step and check if you had predicted the correct results.

3.3 subu

It is a "R" instruction for subtraction. In particular:

```
subu    $t0 , $s0 , $s1
```

does the following: $t0 \leftarrow s0 - s1$.
As we did for `addu`, open **Atom** and create a new file and name it `week8_3_3.asm` as follows:

```
addiu   $s0 , $zero , 1
addiu   $s1 , $zero , −1
addiu   $s2 , $zero , 2
addiu   $s3 , $zero , −2

subu    $t0 , $s0 , $s2
subu    $t1 , $s1 , $s3
subu    $t2 , $s1 , $s2
subu    $t3 , $s0 , $s3
subu    $t4 , $s2 , $s2
subu    $t5 , $s3 , $s3

addiu $v0 , $zero , 10 # to Exit the program
syscall # Exit
```

Codes/week8_3_3.asm

Q.1: Determine what the final value of the following registers will be (interpreted as signed integer)?

t0 :	ffffffff	**t1** :	1
t2 :	fffffffd	**t3** :	3
t4 :	0	**t5** :	0

Execute the instructions step-by-step and check against your prediction.
For the next questions of this part, you do not need to type the requested code and execute it: just write it down here.
Suppose we have loaded some integer values in registers **`s0`**, **`s1`** and **`s2`**. Write a series of instructions (only using the instructions **`addui`**, **`addu`** and **`subu`**) that leads to the result of the requested operation stored in register **`s3`**. To make things even more interesting, try to use at most only one temporary register (e.g. **`t0`**).
Example: for `s3=2*s0`, you can write: `addu $s3, $s0, $s0`.

Q.2: `s3 = s0 + s1 + s2`

```
addu $s3, $s0, $s1, $s2
```

Q.3: `s3 = - s0 - s1`

```
subu $s3, $s0, $s1
```

Q.4: `s3 = s0 - (s1 + s2)`

```
subu $s0, $s1, $s2
```

Q.5: `s3 = 3*s0 - s1`

```
addu $s3, $s0, $s0, $s0, $s1
```

4 Logic Operations

Besides arithmetic operations, the ALU of our MIPS also performs logical operations: Bitwise AND, OR, XOR, NOR.

Q.1: Based on what you learned from the arithmetic instructions, specify the format of the instruction ("I" or "R") and write a brief description for what each of these instructions do:

instruction :	format type	description
ori :	I	It is an operation that between the content of a register and the binary representation of the immediate value.
or :	R	bitwise: OR e.g. or $1, $s2, $s3 -> $s1 = $s2 | $s3 the 'or' value of S2 and s3 are stored in s1.

Q.2: Suppose we have loaded a number in register **s0**. After executing the instruction `andi $t0, $s0, 0x0001` what does the register **t0** hold?
Hint 1: Note that the microprocessor can perform bitwise `and` operation with an integer value, because integers (or anything else for that matter) are represented in binary too.
Hint 2: In our code, we specify the hex values by starting them with a `0x`, so `0x0001` just denotes a 16 bit binary of 15 zeros and a 1 as its lest significant bit (LSB).

andi $t0, $s0, 0x0001
001100 00000 01000 0000000000000001
entire 32 bits: 0011 0000 0000 1000 0000 0000 0000 0001
Hex: 30080001

Q.3: Suppose we have loaded two different values in registers **s0** and **s1**. What will be the content of register **t0** after executing the following instructions?

```
xor     $t0 , $s0 , $s1
xor     $t0 , $t0 , $s1
```

Register t0 is be 0.