

RESEARCH PROJECT SUBMISSION

Architecture Final Assessment

**Program: CESS**

***Course Code: CSE 112***

***Course Name: Computer Organization and Architecture***

***Examination Committee***

**Dr. Tamer Mostafa**

**Ain Shams University**

**Faculty of Engineering**

**Spring 2020 Semester**

**Student Personal Information**

Students’ Names & IDs: Sherif Ahmed Naiem Mohamed 18P6546

Yomna Hussein Mohamed Abd El Hamid 18P5794

Omar Mohamed Lotfy EL-Sayed 18P5606

Nour El-Din Mohamed Hesham 18P5806

Reda Mohsen Reda 18P5141

**Plagiarism Statement**

I certify that this assignment / report is my own work, based on my personal study and/or research and that I have acknowledged all material and sources used in its preparation, whether they are books, articles, reports, lecture notes, and any other kind of document, electronic or personal communication. I also certify that this assignment / report has not been previously been submitted for assessment for another course. I certify that I have not copied in part or whole or otherwise plagiarized the work of other students and / or persons.

**Sherif Ahmed/ Yomna Hussein/ Omar Lotfy/ Nour El-Din Mohamed/ Reda Mohsen**

**Signature/Student Name:**

**3-6-2020**

**Date:**

**Submission Contents**

**01: Introduction to MIPS Code.**

**02: Registers.**

**03: Types of Instructions.**

**04: Instruction Formats.**

**05: Coding & Testing.**

**01**

***First Topic***

**Table of Contents**

[I. Roles Distribution 4](#_Toc41805687)

[II. Introduction to MIPS code: 5](#_Toc41805688)

[a) What is Assembly Language? 5](#_Toc41805689)

[III. Register: 5](#_Toc41805690)

[IV. Types of instruction: 6](#_Toc41805691)

[b) ADD: 6](#_Toc41805692)

[c) ADDI: 7](#_Toc41805693)

[d) SUB: 7](#_Toc41805694)

[e) LW: 8](#_Toc41805695)

[f) SW: 8](#_Toc41805696)

[g) LUI: 9](#_Toc41805697)

[h) AND: 9](#_Toc41805698)

[i) ANDI: 9](#_Toc41805699)

[j) OR: 10](#_Toc41805700)

[k) ORI: 10](#_Toc41805701)

[l) SLL: 11](#_Toc41805702)

[m) SRL: 11](#_Toc41805703)

[n) SRA: 12](#_Toc41805704)

[o) SLT: 12](#_Toc41805705)

[p) SLTI: 13](#_Toc41805706)

[q) NOR: 13](#_Toc41805707)

[r) BEQ: 14](#_Toc41805708)

[s) BNE: 14](#_Toc41805709)

[t) J: 14](#_Toc41805710)

[V. Instruction Formats: 15](#_Toc41805711)

[u) R-Format: 15](#_Toc41805712)

[v) I-Format: 15](#_Toc41805713)

[w) J-Format: 15](#_Toc41805714)

[VI. List of References 16](#_Toc41805715)

# Roles Distribution

1. Yomna Hussien Mohamed

* In Report: Section 3.1 till Section 3.6
* In Coding: I & J Format , BEQ execution code

1. Sherif Ahmed Naiem

* In Report: Section 3.7 till Section 3.12
* In Coding: I & J Format, J execution code

1. Omar Mohamed Lotfy

* In Report: Section 3.13 till Section 3.19
* In Coding: I & J Format, SLTI execution code

1. Nour El-Din Mohamed

* In Report: Introduction , Section 4.2, Section 4.3
* In Coding: Class handling, Class Operations, R Format, Add execution code

1. Reda Mohsen

* In Report: Registers, Section 4.1
* In Coding: Class instructions, Class print, R Format

**Note:** All the team first discussed the project carefully and then we divided the work among us and then after each team member had done his part we all gathered these parts and combined them together to produce this code that translate and execute MIPS instructions code.

# Introduction to MIPS code:

MIPS low level computing language (Assembly) basically alludes to the assembly language of the MIPS processor. The term MIPS is an abbreviation which represents Microprocessor without Interlocked Pipeline Stages, and it is a diminished operation set engineering architecture which was created by an association called MIPS Technologies.

MIPS is a load/store design; Load/store especially used to get to memory, all guidelines and instructions work on the registers.

Those computer instructions are represented in form of bits. Actually, this is the lowest possible level of representation for a program. Each instruction is equivalent to a single action of the CPU. In order that the machine understands this language, it is called machine language [1].

## What is Assembly Language?

A more significant level portrayal and one that is a lot simpler for people to utilize is called Assembly language. Assembly Language firmly associated with machine language, and there is normally a direct relationship to interpret programs written in Assembly into an executable and practical machine code by a [utility program](https://en.wikipedia.org/wiki/Utility_software) referred to as an [assembler](https://en.wikipedia.org/wiki/Assembly_language#Assembler)/compiler. The conversion procedure is alluded to assembly, as in gathering the source code. Assembly language as a rule has one line of code per machine instruction (1:1), yet remarks and explanations that are assembler directives, macros, and significant labels of program and memory areas are frequently additionally boosted [4].

# Register:

A processor register is one of a little set of information holding places that are a piece of the PC processor. A register may hold an instruction, a capacity address, or any sort of information, (for example, a sequence of bits or individual characters). A few guidelines determine enrolls as a component of the instructions. For instance, an instruction may determine that the contents of two characterized registers be included and afterward positioned in a predefined register.

A register must be sufficiently large to hold a guidance - for example, in a [32-bit computer](https://searchdatacenter.techtarget.com/definition/64-bit-processor), a register must be 32 [bits](https://whatis.techtarget.com/definition/bit-binary-digit) in length. In some PC designs, there are smaller registers - for instance, half-registers - for shorter guidelines and instructions. According to the processor design and language rules, registers might be numbered or have discretionary names. A processor normally contains numerous index registers, also called address/index registers or registers of adjustment. The specific location of any entity in a PC incorporates the base, index, and relative locations, which are all put away in the index register. A shift register is another sort of registers. Bits enter the shift register toward one side and rise up out of the opposite end. Flip flops, also called bitable entries, store and procedure the information. General Purpose Registers (GPRs) are demonstrated with a dollar sign ($). The words SWORD and UWORD allude to 32-piece marked and 32-piece unsigned data types, respectively [2].

# Types of instruction:

Instruction in MIPS may vary according to the aim of the user, such like: ADD, ADDI, SUB, SLL,SRL,SRA, AND, ANDI, OR, ORI, NOR, STL, SLTI, LUI, LW, SW,BEQ,BNE, and J.

The previous instructions are classified according to their aim [1]:

* Arithmetic operations: ADD, ADDI, SUB,
* Logic operations: AND, ANDI, OR, ORI, NOR, SLL, SRL, SRA
* Compare instructions: SLT, SLTI
* Move instructions: LUI
* Load and store: LW, SW
* Branch instruction: BEQ, BNE, J

## ADD:

The main goal of this operation is to add the numbers stored in the addresses of the register entered. It is written as following:

C code:

z = x + y + w + v;

MIPS code:

add z, x, y # x + y → z

add z, z, w # z + w → z

add z, z, v # z + v → z

The 32-bit register contains 32 addresses, each address is named according to its place starting from $zero, $s0, $s1, $s2, $s3,….., $9, $t0, $t1, $t2, $t7….etc.

The code in MIPS is written as following:

add $t0, $t1, $t2

This line of code adds the values stored in the second address ($t1) and the third address ($t2), and then the result is stored in the first address ($t0)

|  |  |
| --- | --- |
| Demonstration: | Adds two registers and stores the result in a register |
| Operation meaning: | $f = $t + $s; advance\_pc (4); |
| Syntax form: | add $f, $t, $s |
| Coding: | 0000 00tt ttts ssss ffff f000 0010 0000 |

## ADDI:

The goal of this operation is to add two values; the first value which is stored in the given address of the register, while the second value is a specific number which is not stored in any address (added in a binary form). That i

s why it is called add immediate.

MIPS Code:

addi $s0, $s0, 6 # add immediate

# $s0 (new value/result) = $s0 (old value) + 6

|  |  |
| --- | --- |
| Demonstration: | Adds a register and a sign-extended immediate value and stores the result in a register |
| Operation meaning: | $s = $t + imm; advance\_pc (4); |
| Syntax form: | addi $s, $t, imm |
| Coding: | 0010 00tt ttts ssss iiii iiii iiii iiii |

## SUB:

It demonstrates the same concept of ADD, but the only difference is that this instruction is used for subtraction and it has no immediate.

MIPS code:

Sub $s0, $s1, $s2 #$s0=$s1-$s2

|  |  |
| --- | --- |
| Demonstration: | Subtracts two registers and stores the result in a register |
| Operation meaning: | $f = $t - $s; advance\_pc (4); |
| Syntax form: | sub $f, $t, $s |
| Coding: | 0000 00tt ttts ssss ffff f000 0010 0010 |

## LW:

It is called Load Word and its main purpose is to copy a word from the memory to the register. It means, load into register RegDest the word contained in the location coming about because of including the substance of register RegSource and the Offset determined. The subsequent source address must be word-adjusted (for example numerous of 4).

Therefore, lw $t1,16($0) means to load in $t1 the contents of the word located at address specified by $s0 plus 16. As $0 is register $zero which will always contain the constant zero, it will load the word located in absolute address 8 into $t1 [1].

MIPS Code:

lw $s0, c ($s1) # Memory [$s1 + c] → $s0

|  |  |
| --- | --- |
| Demonstration: | A word is loaded into a register from the specified address. |
| Operation meaning: | $s = MEM[$t + offset]; advance\_pc (4); |
| Syntax form: | lw $s, offset($t) |
| Coding: | 1000 11tt ttts ssss iiii iiii iiii iiii |

## SW:

SW is just like the LW register, the only difference is that the SW-which is short for Store Word-copies a word from the register directly to the memory.

i.e.:

sw $s0, 8($0) # mem[$0+8] <- $s0

Again $0 is the register holding the memory address which contains zero, 8 an offset and $s0 is the source of the information that will be written in memory.

Note that as opposed to others MIPS directions, the primary operand is the source, not the goal. Most likely this is to implement the way that the location register assumes a comparative job in both instructions, while in lw it is used to compute the memory address of the source of information and in sw the memory destination address [1].

|  |  |
| --- | --- |
| Demonstration: | The contents of $s is stored at the specified address. |
| Operation meaning: | MEM[$t + offset] = $s; advance\_pc (4); |
| Syntax form: | sw $s, offset($t) |
| Coding: | 1010 11tt ttts ssss iiii iiii iiii iiii |

## LUI:

 lui is "load upper immediate", with "upper" meaning the left most 16 bits, and "immediate" meaning that you are giving it a specific value.

MIPS code:

Lui $s0,2048

lui $s00, 2048 is the load upper immediate instruction. The immediate value (2048) is shifted left 16 bits and stored in the $s0 register. The lower 16 bits are zeroes.

|  |  |
| --- | --- |
| Demonstration: | The immediate value is shifted left 16 bits and stored in the register. The lower 16 bits are zeroes. |
| Operation meaning: | $s = (imm << 16); advance\_pc (4); |
| Syntax form: | lui $s, imm |
| Coding: | 0011 11-- ---s ssss iiii iiii iiii iiii |

## AND:

The AND instruction’s concept is the same as the AND gate concept; if any of the bits entered in the specific register address is equivalent to zero, therefore the output of the resulting equivalent bit will be equal to zero regardless to the equivalent bit in the other register’s address, otherwise, if both equivalents bits are equal to one, the resulting output result will be equal to one, then the result is stored in the first register address entered.

MIPS code:

And $s0, $s1, $0 #$s0(1001)=$s1(1011) and $0(0000)

|  |  |
| --- | --- |
| Demonstration: | Bitwise ands two registers and stores the result in a register |
| Operation meaning: | $f= $t & $s; advance\_pc (4); |
| Syntax form: | and $f, $t, $s |
| Coding: | 0000 00tt ttts ssss ffff f000 0010 0100 |

## ANDI:

The same concept as the AND instruction, it only has an immediate which compares the bits assigned in the first register with the bits of an entered value by the user.

i.e.:

andi $s0, $s1, 8 #$s0 (1011)=$s1(1011) andi 8(1111)

|  |  |
| --- | --- |
| Demonstration: | Bitwise ands a register and an immediate value and stores the result in a register |
| Operation meaning: | $s= $t & imm; advance\_pc (4); |
| Syntax form: | andi $s, $t, imm |
| Coding: | 0011 00tt ttts ssss iiii iiii iiii iiii |

## OR:

The OR instruction’s concept is the same as the OR gate concept; if any of the bits entered in the specific register address is equivalent to one, therefore the output of the resulting equivalent bit will be equal to one regardless to the equivalent bit in the other register’s address, otherwise, if both equivalents bits are equal to zero, the resulting output result will be equal to zero, then the result is stored in the first register address entered.

MIPS code:

Or $s0, $s1, $s2 #$s0(1111)=$s1(1011) or $s2(1101)

|  |  |
| --- | --- |
| Demonstration: | Bitwise logical ors two registers and stores the result in a register |
| Operation meaning: | $f = $t | $s; advance\_pc (4); |
| Syntax form: | or $f, $t, $s |
| Coding: | 1. tt ttts ssss ffff f000 0010 0101 |

## ORI:

The same concept as the OR instruction, it only has an immediate which compares the bits assigned in the first register with the bits of an entered value by the user.

i.e.:

Ori $s0, $s1, 7 #$s0 (0111)=$s1(0011) ori 7(0111)

|  |  |
| --- | --- |
| Demonstration: | Bitwise ors a register and an immediate value and stores the result in a register |
| Operation meaning: | $s= $t | imm; advance\_pc (4); |
| Syntax form: | ori $s, $t, imm |
| Coding: | 0011 01tt ttts ssss iiii iiii iiii iiii |

## SLL:

A Shift Left Logic of one position moves each bit in the register to one side by one. The right-most bit is supplanted by a zero piece and the left most bit is discarded.

Shifting by two positions is equivalent to shifting by one position, but the bits are only shifted twice. Shifting by zero positions leaves the entered bits unchanged. Shifting by N-bit design left by N or more positions changes the entire bits to zero.

MIPS code:

sll $t1,$s0,2 # $t1 gets the bits in $s0

# shifted left logical two times

|  |  |
| --- | --- |
| Demonstration: | Shifts a register value left by the shift amount listed in the instruction and places the result in a third register. Zeroes are shifted in. |
| Operation meaning: | $f = $s << k; advance\_pc (4); |
| Syntax form: | sll $f, $s, k |
| Coding: | 0000 00tt ttts ssss ffff fkkk kk00 0000 |

## SRL:

A Shift Right Logic of one position moves each bit in the register to one side by one. The left-most bit is supplanted by a zero piece and the right most bit is discarded.

Shifting by two positions is equivalent to shifting by one position, but the bits are only shifted twice. Shifting by zero positions leaves the entered bits unchanged.

The only disadvantage which might make this instruction not practical enough, is that if a negative number is stored in a register and then shifted, the bits after shifting won’t be the same and would be positive which is not logical.

i.e.:

srl $t3, $t3, 1 #The bits stored in $t3 (0110) will be shifted once to the right (0011)

|  |  |
| --- | --- |
| Demonstration: | Shifts a register value right by the shift amount (shamt) and places the value in the destination register. Zeroes are shifted in. |
| Operation meaning: | $f = $s >> k; advance\_pc (4); |
| Syntax form: | srl $f, $s, k |
| Coding: | 0000 00-- ---s ssss ffff fkkk kk00 0010 |

## SRA:

An instruction which is practical to both signed and unsigned integers, as the signed bits after shifting will remain with their same sign.

i.e.:

sra $t0, $s1, 2 #$t0 <- $s1 shifted right twice.

|  |  |
| --- | --- |
| Demonstration: | Shifts a register value right by the shift amount (shamt) and places the value in the destination register. The sign bit is shifted in. |
| Operation meaning: | $f = $s >> k; advance\_pc (4); |
| Syntax form: | sra $f, $s, k |
| Coding: | 0000 00-- ---s ssss ffff fkkk kk00 0011 |

## SLT:

SLT (Set on Less Than) instruction is responsible for comparing the bits set in both registers entered; whether smaller than, greater than, or equal. Slt in MIPS is used for a particular condition like if one value is less than another value, at that point set the estimation of a specific register. It is also can be used with immediate value (SLTI).

i.e.:

**slt $s0, $s1, $s2 #The value which is stored in $s1 will be directly stored in $s0 if it is less**

**than the value in $s2.**

|  |  |
| --- | --- |
| Demonstration: | If $t is less than $s, $f is set to one. It gets zero otherwise. |
| Operation meaning: | if $t < $s $f = 1; advance\_pc (4); else $f = 0; advance\_pc (4); |
| Syntax form: | slt $f, $t, $s |
| Coding: | 0000 00tt ttts ssss ffff d000 0010 1010 |

## SLTI:

SLTI (Set on less than immediate) has the same instruction concept of SLT, the only difference is that it compares specific value stored in the register with a certain immediate value entered by the user.

i.e.:

**slti $s0, $s1, 8 #The value which is stored in $s1 will be directly stored in $s0 if it is less than 8**

|  |  |
| --- | --- |
| Demonstration: | If $t is less than immediate, $s is set to one. It gets zero otherwise. |
| Operation meaning: | if $t < imm $s = 1; advance\_pc (4); else $s = 0; advance\_pc (4); |
| Syntax form: | slti $s, $t, imm |
| Coding: | 0010 10tt ttts ssss iiii iiii iiii iiii |

**NOR:**

## **NOR:**

**The NOR instruction usage is as the same as the NOR Gate use; it is used like it reverses the role of the or gate. Like we have said before, in the OR gate,** if any of the bits entered in the specific register address is equivalent to one, therefore the output of the resulting equivalent bit will be equal to one regardless to the equivalent bit in the other register’s address, otherwise, if both equivalents bits are equal to zero, the resulting output result will be equal to zero.

The NOR gate reverses all the previous steps; as the resulting equivalent bit will equal to 1 if both bits entered in both registers are equal to 0, otherwise the resulting equivalent bit will be equal to 0.

NB: The second register is always $0 which contains zero inside.

i.e.:

nor $s1, $s2, $s3 #$s1(1000)=$s2(0110) nor $s3(0101)

|  |  |
| --- | --- |
| Demonstration: | Bitwise logical nors two registers and stores the result in a register |
| Operation meaning: | $f = $t |’ $0; advance\_pc (4); |
| Syntax form: | nor $f, $t, $0 |
| Coding: | 0000 00tt ttt0 0000 ffff f000 0010 0101 |

## BEQ:

BEQ (short for "Branch if Equal") is the memory aide for a machine language instruction which branches to the location indicated, and just if the zero banner is set. If the zero banner is clear when the CPU experiences a BEQ instruction, the CPU will proceed at the instruction following the BEQ as it will not take the jump.

i.e.:

beq $s0, $s1, 56 (offset) #if $s0 === $s1, $s1 will jump 56 steps from the offset.

|  |  |
| --- | --- |
| Demonstration: | Branches if the two registers are equal |
| Operation meaning: | if $t == $s advance\_pc (offset << 2)); else advance\_pc (4); |
| Syntax form: | beq $t, $s, offset |
| Coding: | 0001 00tt ttts ssss iiii iiii iiii iiii |

## BNE:

BNE (short for Branch on Not Equal); it has got the same concept of the BEQ, but instead, it branches if the two registers are not equal.

i.e.:

bne $s0, $s1, 56 (offset) #if $s0 != $s1, $s1 will jump 56 steps from the offset.

|  |  |
| --- | --- |
| Demonstration: | Branches if the two registers are not equal |
| Operation meaning: | if $t != $s advance\_pc (offset << 2)); else advance\_pc (4); |
| Syntax form: | bne $t, $s, offset |
| Coding: | 0001 01tt ttts ssss iiii iiii iiii iiii |

## J:

The J instruction loads a value immediately to the computer register. This value might be an offset or may be a label (the assembler converts the label into an offset). This instruction need a 26-bit coded address field to determine and calculate the jump target, also the other 6-bit coded address field is for the opcode.

It is a type of branching instructions.

i.e.:

j exit #label is a word address.

|  |  |
| --- | --- |
| Demonstration: | Jumps to the determined address |
| Operation meaning: | PC = nPC; nPC = (PC & 0xf0000000) | (target << 2); |
| Syntax form: | J target |
| Coding: | 0000 10ii iiii iiii iiii iiii iiii iiii |

# Instruction Formats:

Instruction format is a format which describes the layout of a certain instruction, in terms of divided parts. It includes opcode (6 bits), and the address dependent factors (26 bits).

Instruction Formats are divided into 4 essential formats, each format is responsible for certain instructions which are [3]:

## R-Format:

It includes op which is always zero (6 bits), rs which is the argument of the 1st register (5 bits), rt which is the argument of the 2nd register (5 bits), rd which is the destination register (5 bits), shamt-which is used only in case of shift instructions-(5 bits), func which represents the code for the operation to form (6 bits)🡪 32 bits in total.

*R-Format Instructions:* ADD, AND, NOR, OR, SLL, SLT, SRL, SRA [3].

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | op | rs | rt | rd | shamt | Func. |
| R-Format | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

## I-Format:

It includes op (6 bits), rs which is the argument of the 1st register (5 bits), rt which is the argument of the 2nd register (5 bits), and immediate address (16 bits) 🡪 32 bits in total.

*I-Format Instructions:* ADDI, ANDI, BEQ, BNE, LUI, LW, ORI, SLTI, SW [3].

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | op | rs | rt | Immediate Address |
| I-Format | 6 bits | 5 bits | 5 bits | 16 bits |

## J-Format:

Includes op (6 bits), and a label address (26 bits) 🡪 32 bits in total

*J-Format Instructions:* J

|  |  |  |
| --- | --- | --- |
|  | op | Target address |
| J-Format | 6 bits | 26 bits |

# List of References

[1] Vollmar, K., & Sanderson, P. (2006, March). MARS: an education-oriented MIPS assembly language simulator. In *Proceedings of the 37th SIGCSE technical symposium on Computer science education* (pp. 239-243)

[2] Flynn, M. J. (1995). *Computer architecture: Pipelined and parallel processor design*. Jones & Bartlett Learning.

[3] Gupta, S. A., Rau, B. R., Johnson, R. C., & Schlansker, M. S. (2002). *U.S. Patent No. 6,457,173*. Washington, DC: U.S. Patent and Trademark Office.

[4] Morrisett, G., Walker, D., Crary, K., & Glew, N. (1999). From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *21*(3), 527-568.

.