

MAJOR COMPUTER SCIENCE

START TIME AND DATE: 10/28/2017 11:00 AM

END TIME AND DATE: 10/29/2017 7:00 AM

TAKE HOME TEST 1

CSC 343 Fall 2017

October 30, 2017

Jeter Gutierrez

TABLE OF CONTENTS**1. Objective pg 4****PART 1 : MIPS ANALYSIS pg 5-28****2.1 Variables In Mips. 5-10****2.2 While loops in MIPS pg 10-13****2.3 For loops in MIPS pg 13-17****2.4 If-Then-Else statements in MIPS pg 17-20****2.5 Calling myadd Function in MIPS pg 20-23****2.6 Logical Operators in MIPS pg 23-26****2.7 Using arrays in MIPS pg 26-28****PART 2 LINUX 64 BIT COMPILER pg 28-41****3.1 Local variables in Linux 64 Bit Compiler pg 28-30****3.2 Static variables in Linux 64 Bit Compiler pg 30-31****3.3 While Loop in Linux 64 Bit Compiler pg 31-32****3.4 For Loop in Linux 64 Bit Compiler pg 32-34****3.5 If-then-else statements in Linux 64 Bit Compiler pg 34-36****3.6 Calling Myadd Function in Linux 64 Bit Compiler pg 36-38****3.7 Logical Operators in Linux 64 Bit Compiler pg 38-40****3.8 Using arrays in Linux 64 Bit Compiler pg 40-41****PART 3 LINUX 32 BIT ON RASPBERRY PI pg 41-58****4.1 Local variables on Raspberry Pi pg 41-44**

4.2 Static variables on Raspberry Pi pg 44-45

4.3 While Loops on Raspberry Pi pg 45-47

4.4 For Loops on Raspberry Pi pg 47-49

4.5 If-then-else statements on Raspberry Pi pg 49-52

4.6 Calling Myadd Function on Raspberry Pi pg 52-54

4.7 Logical Operators on Raspberry Pi pg 54-56

4.8 Using Array on Raspberry Pi pg 56-58

PART 4 WINDOWS 32 BIT COMPILER pg 58-82

5.1 Local Variables on Windows 32 Bit Compiler pg 58-61

5.2 Static Variables on Windows 32 Bit Compiler pg 61-65

5.3 While Loop on Windows 32 Bit Compiler pg 65-68

5.4 For Loop on Windows 32 Bit Compiler pg 68-71

5.5 If-then-else statements on Windows 32 Bit Compiler pg 71-73

5.6 Calling myadd function on Windows 32 Bit Compiler pg 73-76

5.7 Logical Operators on Windows 32 Bit Compiler pg 76-79

5.8 Using arrays on Windows 32 Bit Compiler pg 79-82

6 Conclusion pg 82

1. Objective

The purpose of this test is for us to demonstrate how well we understand MIPS instruction set architecture, Intel x86 ISA using Windows MS 32-bit compiler and debugger, and an Intel X86_64 bit ISA processor running Linux, 64 bit gcc and gdb, Cortex v7 instruction set architecture on Raspberry Pi platform. We will need to compare the differences between the 4 architectures and properly understand what they share in common too. For us to properly compare the 4 ISA's we will first have an understanding of the sections 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, and 2.8 on the textbook **Computer Organization and Design**. In order to test the MIPS processor we will be running the programs on a MARS MIPS simulator that we can find online. In order to test Intel X32 ISA, Windows 32-bit compiler we will be running programs on Visual Studio and using the softwares debugger. For testing of X86_64 ISA we will be running 64bit GCC and GDB on a Linux 64-bit platform. We will use a Raspberry pi to test Cortex v7 ISA. For each Processor we will be testing the following in examples in order to understand how each processor works and how they differ from each other:

1. Using Local Variables.
2. Using Static Variables.
3. While loops.
4. For loops.
5. If-then-else loops.
6. Calling a function other than main.
7. Logical Operators.

8. Using arrays.

PART 1 : MIPS ANALYSIS

In this section we will be testing MIPS instructions and observing how the compiler for the software works. We will be using assembly code for MIPS processor on MARS and then we will be observing what is happening in memory as we run through the program. We will be observing how MIPS interacts and handles Variables, while loops, for loops, if-then-else statements, calls a function, uses logical operators, and accesses an array. We will be breaking down what is happening in each instance and then we will be able to continue with the project onto 3 other platforms which will be running C code and will be compared to the code written here in assembly for MIPS. We will also note that MIPS uses little Endian, that means it stores the least significant byte of information in memory into the lowest address.

2.1 Variables In Mips.

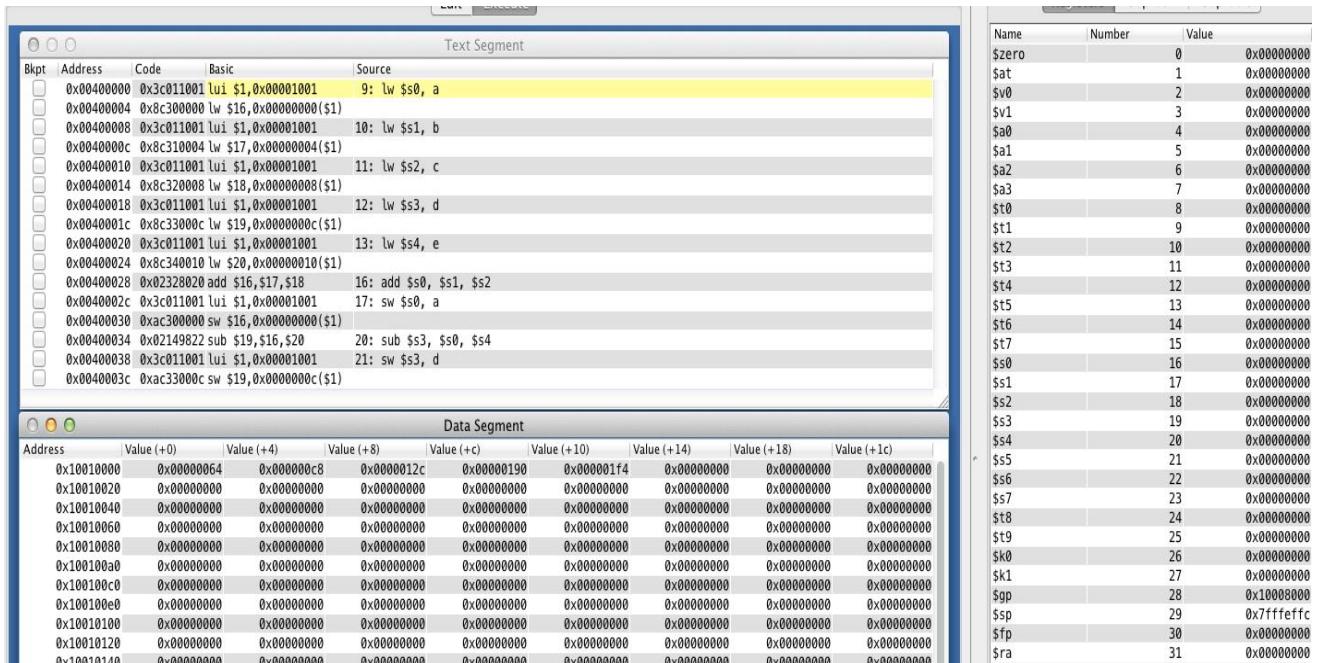
In this section we will be observing how MIP ISA uses variables, how it stores them and what happens when they are being used.

```

1 .data
2 a: .word 100
3 b: .word 200
4 c: .word 300
5 d: .word 400
6 e: .word 500
7
8 .text
9 lw $s0, a
10 lw $s1, b
11 lw $s2, c
12 lw $s3, d
13 lw $s4, e
14
15 # a = b + c
16 add $s0, $s1, $s2
17 sw $s0, a
18
19 # d = a - e
20 sub $s3, $s0, $s4
21 sw $s3, d

```

Figure 1: Assembly code for MIPS addition and subtraction using 5 local variables in memory.



The screenshot shows a debugger interface with two main windows: 'Text Segment' and 'Registers'.

Text Segment: This window displays the assembly code with line numbers and comments. The comments indicate the purpose of each instruction: loading variables into registers (\$s0-\$s4), performing addition and subtraction (\$s0 + \$s1 - \$s4), and storing the result back into memory at address \$s3.

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001	lui \$1,0x00001001	9: lw \$s0, a
	0x00400004	0xb8c30000	lw \$16,0x0000000(\$1)	
	0x00400008	0x3c011001	lui \$1,0x00001001	10: lw \$s1, b
	0x0040000c	0xb8c31004	lw \$17,0x00000004(\$1)	
	0x00400010	0x3c011001	lui \$1,0x00001001	11: lw \$s2, c
	0x00400014	0xb8c32008	lw \$18,0x00000008(\$1)	
	0x00400018	0x3c011001	lui \$1,0x00001001	12: lw \$s3, d
	0x0040001c	0xb8c3300c	lw \$19,0x0000000c(\$1)	
	0x00400020	0x3c011001	lui \$1,0x00001001	13: lw \$s4, e
	0x00400024	0xb8c34001	lw \$20,0x00000010(\$1)	
	0x00400028	0xb2328020	add \$16,\$17,\$18	16: add \$s0, \$s1, \$s2
	0x0040002c	0x3c011001	lui \$1,0x00001001	17: sw \$s0, a
	0x00400030	0xa0c30000	sw \$16,0x00000000(\$1)	
	0x00400034	0xb02149822	sub \$19,\$16,\$20	20: sub \$s3, \$s0, \$s4
	0x00400038	0x3c011001	lui \$1,0x00001001	21: sw \$s3, d
	0x0040003c	0xa0c33000c	sw \$19,0x0000000c(\$1)	

Data Segment: This window shows memory dump starting at address 0x10010000. It displays values for memory locations from \$s0 to \$s4.

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000064	0x000000c8	0x0000012c	0x00000190	0x000001f4	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Registers: This window shows the state of the processor registers. The stack pointer (\$sp) is currently at 0x7ffffeffc.

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7ffffeffc
\$fp	30	0x00000000
\$ra	31	0x00000000

Figure 2: Compiled code of MIPS processor using local variables to perform addition and subtraction. As we can see in the registers window there is information stored by default already in the \$sp which is the address of the stack pointer, it is currently 0x7ffffeffc this is because by

default it has to assign the proper address to the stack pointer before being able to perform any other operation at all. In the addresses as we can see in the data segment addresses 0x10010000 to 0x10010010 we have information already stored in these addresses due to the use of static variables in our Mips assembly. In the address 0x10010000 which is the address of variable a the value stored there is 100 in decimal and 64 in hexadecimal format. This extends to variable b, c, d and e in the same manner all the way to 1f4 which is the value of 500.

In the registers window we can see the address of the stack pointer but we can also see the address of pc which is the address of the next instruction as we can see it is 0x00400000 both in pc and in the first address in the text segment. The global pointer has an address of 0x10080000 which is half way from the instruction pointer and the stack pointer.

Let's move a few steps forward and see what is happening in the compiler.

The screenshot shows a debugger interface with three main windows:

- Text Segment:** Displays assembly code with columns for Bkpt, Address, Code, Basic, and Source. The code includes instructions like `lui`, `lw`, `add`, and `sw`. Line 6 is highlighted in yellow.
- Data Segment:** A table showing memory addresses from 0x10010000 to 0x10010180. Column headers include Address, Value (+0), Value (+4), Value (+8), Value (+c), Value (+10), Value (+14), Value (+18), and Value (+1c). The value for address 0x10010000 is 0x00000064.
- Registers:** A table showing register names, numbers, and values. The \$at register is highlighted in green, showing a value of 1 (0x10010000).

Figure 3: as we can see after i step into the code up to line 6 if we look at the register \$at which is the assembly temporary register we can see that the address in memory is 0x10010000 and the offset as explained from `lw $18,0x00000008($1)` is 8, this means that what it is going to do now

is go to the address data segment window and store what is in the address 0x10010008 into the register \$s2. And as we can see in register \$s0 the value of variable 1 is stored which is the same as what is in the address 0x1001000 and that in \$s2 the value of variable b is stored as is expected through the instructions that were sent to the processor.

If we continue stepping into the program we can see what happens when we are adding in mips and when we are subtracting.

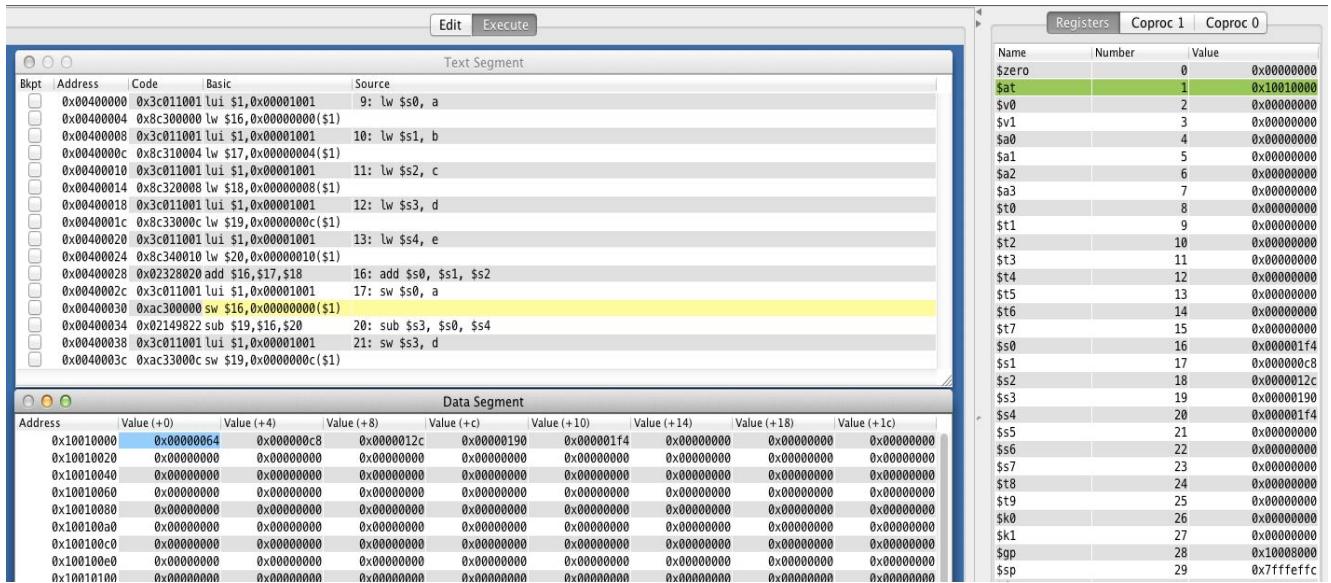


Figure 4: MIPS registers and text segment and data segment after running the add command.

The compiler has correctly added the values from \$s1 and \$s2 into register \$s0 it has added 200 and 300 and stored its value 500 into register \$s0 by using the \$at temporary register each time it was performing a task. Now the value of a is also 500.

As we can see the value in register \$s0 and \$s4 are the same, they are both 500, after we run subtraction in order for the mips processor to be reacting as expected we are going to expect for there to be a value of 0 in register \$s3 after the operation. In the same way that we notice that the

address of register pc will be the same as the final address highlighted in the yellow line as it is usually.

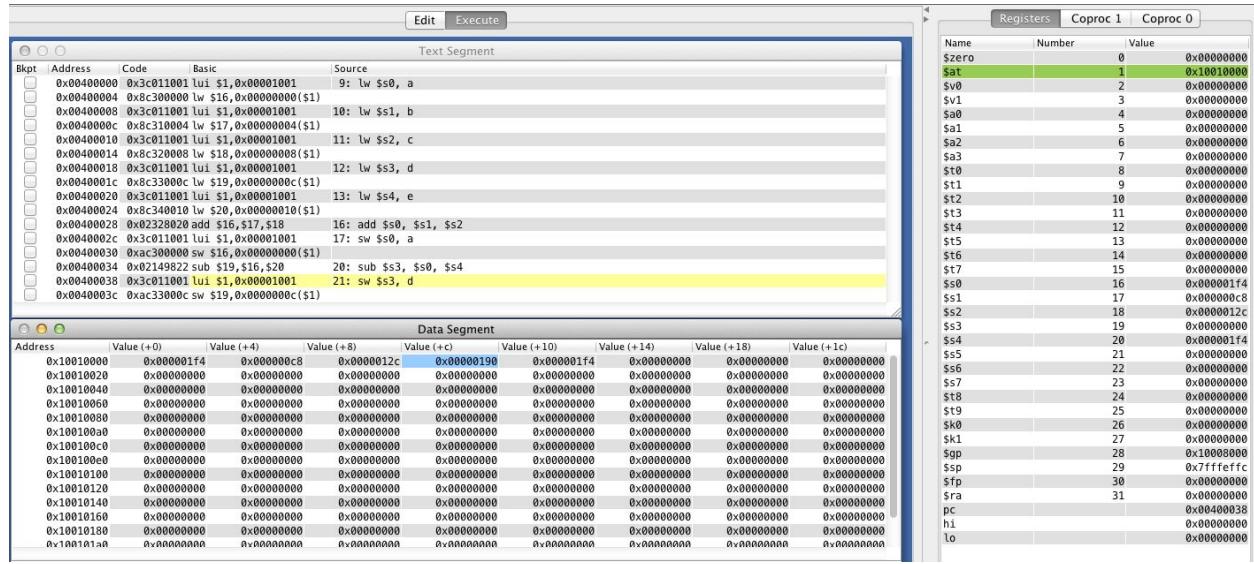


Figure 5: Mips simulator after running subtraction.

As expected the value in register \$s3 is now 0 because we have subtracted 500 and 500 which is what were in the required registers when we were performing the subtraction operation. However if we see in the data segment address 0x1001000c the value is not yet 0 because it will be after we perform the next and final step in the program.

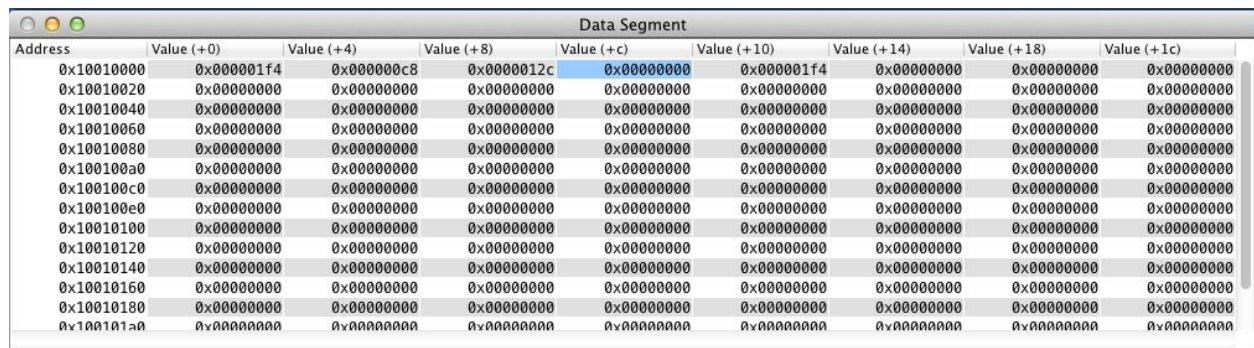


Figure 6: Mips data segment after terminating the program.

Now that it is finished we can see that the value at the address 0x1001000c is also 0 because the subtraction as well as the final line of the program have been successfully performed.

It is important to note that the reason we easily see the value or the information stored in the given address without having to perform any conversion is because MIPS uses little endian which means that the information is stored in the order it is read, it stores the lowest byte of information at the lowest address, this means it is stored in proper reading order.

2.2 While loops in MIPS

In this section we will be testing and observing how a while loop functions in MIPS we will be using this to see the difference between just using a while loop and just using variables.

```
GUTIERREZ_2 WHILE_LOOP.asm
1 .data
2     message: .asciiz "Finished with the While Loop"
3 .text
4     addi $t0, $zero, 0 #Initializes the initial value to 0.
5
6 while: bgt $t0, 10, exit #Compares the values and runs until the value is equal to 10 or less than 10.
7
8     addi $t0,$t0,1 #increases the iterators value by 1 stored in register $t0
9     j while #continue the while loop
10 exit:
11    li $v0,4
12    la $a0, message
13    syscall #End of program
14    li $v0, 10
15    syscall
```

Figure 7: Assembly code for MIPS using a while loop.

In this code we will have an iterator value that will start at 0 and be increasing by 1 while it is less than 10.

The screenshot shows a MIPS assembly debugger interface. The top menu bar includes File, Edit, Run, Settings, Tools, Help, and a toolbar with various icons. A status bar at the bottom indicates "Run speed at max (no interaction)".

Text Segment: This window displays assembly code with comments. The code initializes a value to zero, compares it with 10, increments the value, and prints it before exiting.

```

Text Segment
Blkpt Address Code Basic Source
0x00400000 0x20080000 addi $t0, $zero, 0 #Initializes the initial value to 0.
0x00400004 0x2001000a addi $t1, $0, 0x0000000a 4: addi $t0, $zero, 0 #Initializes the initial value to 0.
0x00400008 0x0028082a slt $t1,$t1,$8 6: while: bgt $t0, 10, exit #Compares the values and runs until the value is equal to 10 or less than 10.
0x00400010 0x21080001 addi $t8,$t8,0x00000001 8: addi $t0,$t0,1 #increases the iterators value by 1 stored in register $t0
0x00400014 0x08010001 j 0x00400004 9: j while #contine the while loop
0x00400018 0x24020004 addiu $t2,$0,0x00000004 11: li $v0,4
0x0040001c 0x3c011001 lui $t1,0x000001001 12: la $a0,message
0x00400020 0x34240000 ori $t4,$1,0x00000000
0x00400024 0x0000000c syscall 13: syscall #End of program
0x00400028 0x2402000a addiu $t2,$0,0x0000000a 14: li $v0,10
0x0040002c 0x0000000c syscall 15: syscall

```

Data Segment: This window shows memory starting at address 0x10010000. It lists addresses and their corresponding values for the .data section.

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x696e6946	0x64656873	0x74697720	0x68742068	0x68572065	0x20656c69	0x706f6f4c	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010160	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Registers: This window shows the state of all MIPS registers (\$zero through \$lo).

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10000000
\$sp	29	0x7ffffeffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

Figure 8: Disassembly window, Data Segment and Registers for MIPS immediately after assembling the while loop assembly code.

Initially nothing has changed, other than the \$gp which is the address of the global pointer and \$sp which is the address of the stack pointer. In the data segment the values of the strings are already stored so that they can be used later in the code when the while loop is finished.

The screenshot shows a MIPS assembly debugger interface with three main windows:

- Text Segment:** Displays assembly code with comments. The code initializes a value to zero, runs a loop until it's less than or equal to 10, increments the value by one each iteration, prints the value, and ends with a syscall.
- Data Segment:** Shows memory dump starting at address 0x10010000, where all memory is currently zeroed.
- Registers:** Shows the state of various MIPS registers. The \$t0 register is highlighted in green, indicating its current value of 1.

```

Text Segment:
Blkpt Address Code Basic Source
0x00400000 0x20080000 addi $8,$0,0x00000000 4: addi $t0,$zero,0 #Initializes the initial value to 0.
0x00400004 0x2001000a addi $1,$0,0x0000000a 6: while: bgt $t0,10,exit #Compares the values and runs until the value is equal to 10 or less than 10.
0x00400008 0x0028082a slt $1,$1,$8
0x0040000c 0x14200002 bne $1,$0,0x00000002
0x00400010 0x21080001 addi $8,$8,0x00000001 8: addi $t0,$t0,1 #increases the iterators value by 1 stored in register $t0
0x00400014 0x08100001 j 0x00400004 9: j while #contine the while loop
0x00400018 0x24020004 addiu $2,$0,0x00000004 11: li $v0,4
0x0040001c 0x3c011001 lui $1,0x00001001 12: la $a0,message
0x00400020 0x34240000 ori $4,$1,0x00000000
0x00400024 0x0000000c syscall 13: syscall #End of program
0x00400028 0x2402000a addiu $2,$0,0x0000000a 14: li $v0,10
0x0040002c 0x0000000c syscall 15: syscall

Data Segment:
Address Value (+0) Value (+4) Value (+8) Value (+c) Value (+10) Value (+14) Value (+18) Value (+1c)
0x10010000 0x696e6946 0x64656873 0x74697720 0x68742068 0x68572065 0x20656c69 0x706f6f4c 0x00000000
0x10010020 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x10010040 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x10010060 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x10010080 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x100100a0 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x100100c0 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x100100e0 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x10010100 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x10010120 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x10010140 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x10010160 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000

Registers:
Name Number Value
$zero 0 0x00000000
$at 1 0x00000000
$v0 2 0x00000000
$v1 3 0x00000000
$a0 4 0x00000000
$a1 5 0x00000000
$a2 6 0x00000000
$a3 7 0x00000000
$t0 8 0x00000001
$t1 9 0x00000000
$t2 10 0x00000000
$t3 11 0x00000000
$t4 12 0x00000000
$t5 13 0x00000000
$t6 14 0x00000000
$t7 15 0x00000000
$s0 16 0x00000000
$s1 17 0x00000000
$s2 18 0x00000000
$s3 19 0x00000000
$s4 20 0x00000000
$s5 21 0x00000000
$s6 22 0x00000000
$s7 23 0x00000000
$t8 24 0x00000000
$t9 25 0x00000000
$k0 26 0x00000000
$k1 27 0x00000000
$gp 28 0x10000000
$sp 29 0xffffffff
$fp 30 0x00000000
$ra 31 0x00000000
pc 0x00400010
hi 0x00000000
lo 0x00000000

```

Figure 9: Disassembly window, Data Segment and Registers for MIPS immediately after one iteration.

As we can see the value stored in \$t0 is equal to 1 because we have gone through the program one step already, now it will check if the value continues to be less than 10 using bgt and it will continue to increase the value stored in \$t0 which is our iterator value.

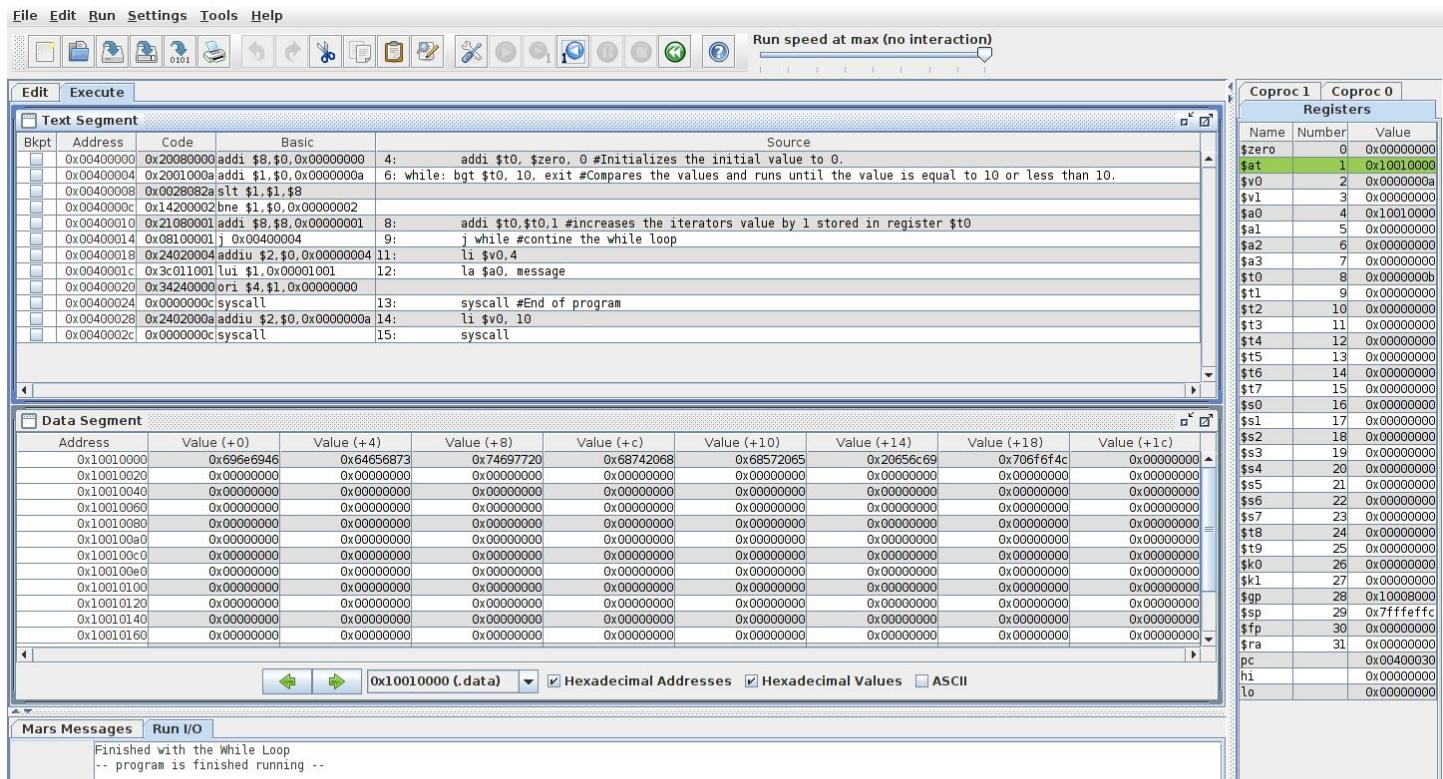


Figure 10: Final image of running a while loop on MIPS.

After running the while loop on MIPS we can see that the value of the counter is now greater than 10 it is 0x0000000b or 11 the reason for this is because that is allowed, now that the value is greater than 10 it will not iterate back into the while loop but it does not give any errors, instead it will read what is stored in the data segments corresponding to the string values and print in the bottom of the run windows that it is finished with the while loop. As we can see the value stored in \$at now is 0x10010000 which is the head of the string that is being printed which starts with the hexadecimal value stored in the first data segment as 0x696e6946 which is part of the string value.

2.3 For loops in MIPS

In this section we will be looking at how MIPS interacts with for loops and what happens with the memory and how we can test for loops more effectively.

```
GUTIERREZ_3_FOR_LOOPS.asm
1 li $t0, 15 # t0 is a constant ending value to exit the for loop
2 li $t1, 0 # initialize the counter to 0
3 forLoop:
4 beq $t1, $t0, end # for int i=0, i<10 i++
5 addi $t2, $t2, 100 #the body of the loop, what else is happening besides checking the condition on the following line.
6 addi $t1, $t1, 1 # add 1 to t1 increments the counter.
7 j forLoop # jump back to the top goes back to the top of the loop until condition is met
8 end:
```

Figure 11: Assembly code for MIPS for loop.

In this code we will be looking at a for loop that will increment the value at the register \$t1 which will initially be 0 and we will be comparing it to the value stored in \$t0 which will constantly be 15, the reason for this is because we will be testing that for as long as \$t1 is less than 15 we will increment it by 1 and then we will also be adding 100 to the register \$t2 just to have a body of the for loop. J here is used similar to a function call, it is returning to the address where the for loop is and then comparing the statement again, until completion. It will run until the value at the register \$t0 is equal to the value at the register \$t1.

The screenshot shows a MIPS assembly debugger interface. The top menu bar includes File, Edit, Run, Settings, Tools, and Help. Below the menu is a toolbar with various icons for file operations and simulation controls. A status bar at the bottom indicates "Run speed at max (no interaction)".

Text Segment: This window displays the assembly code for a for loop. The code includes instructions for initializing registers \$t0 and \$t1 to 0, setting up a loop condition (\$t0 <= 15), and incrementing \$t1 by 1 in each iteration. The assembly code is as follows:

```

Basic
1: li $t0, 15 # t0 is a constant ending value to exit the for loop
2: li $t1, 0 # initialize the counter to 0
3: beq $t1, $t0, end # for int i=0, i<10 i++
4: addi $t2, $t2, 100 #the body of the loop, what else is happening besides checking the condition on the followin...
5: addi $t2, $t2, 100
6: addi $t1, $t1, 1 # add 1 to t1 increments the counter.
7: j forLoop # jump back to the top goes back to the top of the loop until condition is met

```

Data Segment: This window shows memory starting at address 0x10010000. All memory cells from address 0x10010000 to 0x10010160 contain the value 0x00000000.

Registers: This window lists the 32 general-purpose registers (\$zero through \$t6, \$s0 through \$s2, \$s3 through \$s7, \$t8 through \$t9, \$k0 through \$k1, \$gp through \$fp, \$ra, pc, hi, and lo) along with their current values. Register \$t0 is set to 0x00000000.

Figure 12: Disassembly window, Data Segment and Registers for MIPS immediately after assembling the For Loop code.

At this point the value stored in register \$t0 is set to be 0 it will be the value that we are iterating in comparison to. We will be increasing \$t1 until reaching 15 in our for loop.

The screenshot shows a MIPS assembly debugger interface. The top menu bar includes File, Edit, Run, Settings, Tools, and Help. Below the menu is a toolbar with various icons for file operations and simulation controls. A progress bar labeled "Run speed at max (no interaction)" is located at the top right.

Text Segment:

Blkpt	Address	Code	Basic	Source
0x00400000	0x2408000f	addiu \$8, \$0, 0x0000000f	1: li \$t0, 15 # t0 is a constant ending value to exit the for loop	1: li \$t0, 15 # t0 is a constant ending value to exit the for loop
0x00400004	0x24090000	addiu \$9, \$0, 0x00000000	2: li \$t1, 0 # initialize the counter to 0	2: li \$t1, 0 # initialize the counter to 0
0x00400008	0x11280003	beq \$9,\$8,0x00000003	4: beg \$t1, \$t0, end # for int i=0, i<10 i++	4: beg \$t1, \$t0, end # for int i=0, i<10 i++
0x0040000c	0x214a0064	addi \$10,\$10,0x0000...5:	5: addi \$t2, \$t2, 100 #the body of the loop, what else is happening besides checking the condition on the followin...	5: addi \$t2, \$t2, 100 #the body of the loop, what else is happening besides checking the condition on the followin...
0x00400010	0x21290001	addi \$9,\$9,0x00000001	6: addi \$t1, \$t1, 1 # add 1 to t1 increments the counter.	6: addi \$t1, \$t1, 1 # add 1 to t1 increments the counter.
0x00400014	0x08100002	j 0x00400008	7: j forLoop # jump back to the top goes back to the top of the loop until condition is met	7: j forLoop # jump back to the top goes back to the top of the loop until condition is met

Data Segment:

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010160	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Registers:

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x0000000f
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10000000
\$sp	29	0x7ffffeffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400004
hi		0x00000000
lo		0x00000000

Figure 13: Disassembly window, Data Segment and Registers for MIPS immediately after

setting the value at \$t0 to 15.

This screenshot is identical to Figure 13, showing the same assembly code, data segment, and register values. The only difference is that the instruction at address 0x00400014 (j 0x00400008) is highlighted in yellow, indicating it is the current instruction being executed or selected.

Figure 14: Disassembly window, Data Segment and Registers for MIPS immediately after setting our iterators value to 1. After the first iteration.

We will not skip ahead a few steps into the iteration to show that as we continue to iterate through the for loop the value stored in register \$t1 is increasing by 1 and the value stored in \$t2 is increasing by 100. The value increasing by 100 is not being compared to anything at all it is just a value that is being changed in the body of our for loop to show that we can perform more than one task at a time.

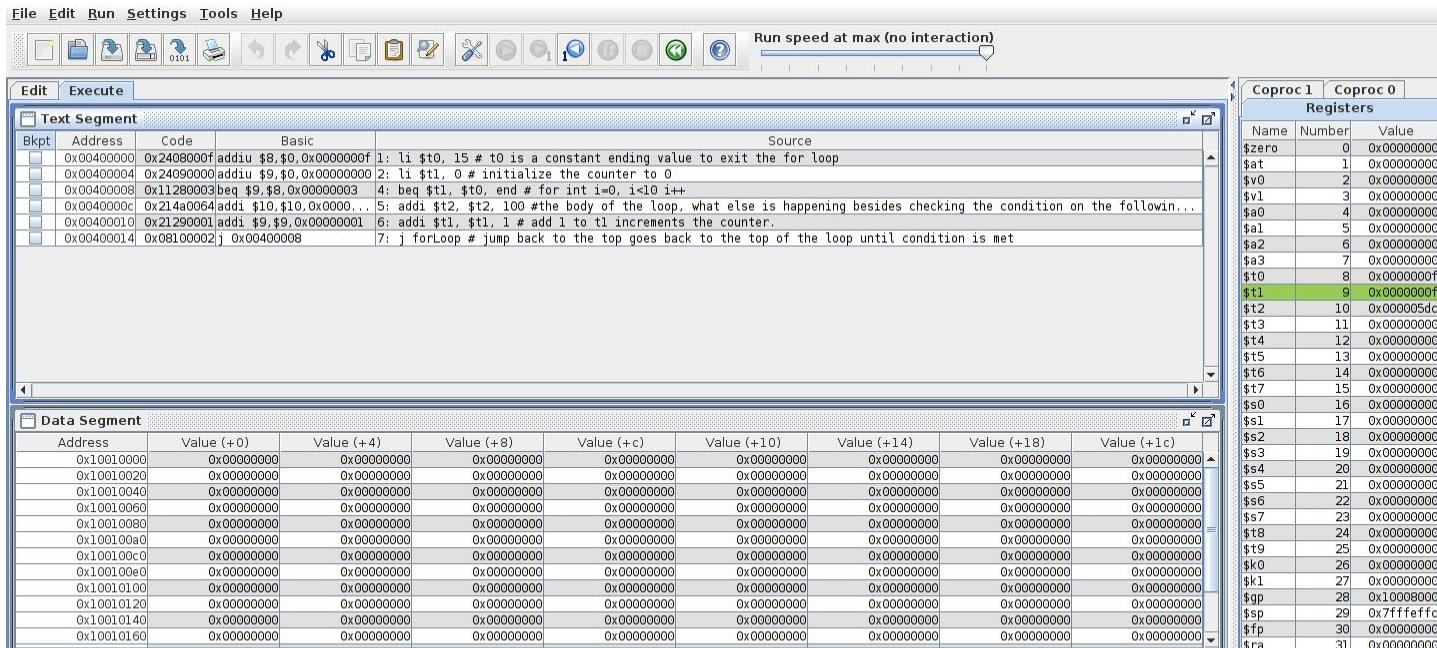


Figure 15: Disassembly window, Data Segment and Registers for MIPS after finalizing the for loop.

At the end of our for loop the value stored in register \$t1 is equal to 15 which is the same value stored in register \$t0 which is why our for loop is finished.

2.4 If-Then-Else statements in MIPS

We will be testing if-then-else statements now in MIPS.

```

.data
    DIFFERENT: .asciiz "DIFFERENT."
    SAME: .asciiz "EQUAL."
.text

main:
    addi $t0, $zero, 20#First number
    addi $t1, $zero, 20#Second Number
if:
    beq $t0,$t1 numbersEqual#condition only true if the numbers being compared are equal.
else:
    bne $t0,$t1 numbersDifferent #true if the numbers being compared are not equal.
    li $v0, 10
    syscall

numbersDifferent:
    li $v0, 4
    la $a0, DIFFERENT #Loads text into register to print
    syscall
    li $v0, 10
    syscall

numbersEqual:
    li $v0, 4
    la $a0, SAME #Loads text into register to print.
    syscall
    li $v0, 10
    syscall

```

[Figure 16: Assembly code for if-then-else statements for MIPS.](#)

We will be using strings here in order to compare values and use our if then else statements properly while seeing the change outside of the program through the console. The strings are “Different” and “Equal” in order to represent if the two numbers 20 stored in \$t0 and 20 stored in \$t1 are equal or different. We logically know that they are equal we want to check that MIPS will return the correct output. Beq means branch if equal, and bne means branch if not equal. If this works as expected nothing will happen in the first bqe but it will make a function call to numbersEqual in beq statement.

The screenshot shows the MIPS Simulator interface. The top part displays the assembly code in the Text Segment window:

```

Edit Execute
Text Segment
Bkpt Address Code Basic Source
0x00400004 0x20090014 addi $8,$0,0x00000014 8: addi $t0,$zero,20#change these numbers to fall into each of the different conditions
0x00400008 0x10900009 beg $8,$9,0x00000009 9: addi $t1,$zero,20#can also change this number
0x0040000c 0x15090002 bne $8,$9,0x00000002 10: if: beg $t0,$t1 numbersEqual#if the two numbers are equal it will print the numbers are equal
0x00400010 0x2402000a addiu $2,$0,0x0000000a 12: else: bne $t0,$t1 numbersDifferent #else if the numbers are not equal it will print that the numbers are diff...
0x00400014 0x0000000c syscall 13: li $v0,10
0x00400018 0x24020004 addiu $2,$0,0x00000004 14: syscall
0x0040001c 0x3c01001 lui $1,0x00001001 17: li $v0,4
0x00400020 0x34240000 ori $4,$1,0x0000002d 18: la $a0,message
0x00400024 0x0000000c syscall 19: syscall
0x00400028 0x2402000a addiu $2,$0,0x0000000a 20: li $v0,10
0x0040002c 0x0000000c syscall 21: syscall
0x00400030 0x24020004 addiu $2,$0,0x00000004 24: li $v0,4
0x00400034 0x3c01001 lui $1,0x00001001 25: la $a0,message3
0x00400038 0x34240002 ori $4,$1,0x0000002d
0x0040003c 0x0000000c syscall 26: syscall
0x00400040 0x2402000a addiu $2,$0,0x0000000a 27: li $v0,10

```

The Data Segment window shows the memory dump at address 0x10010000:

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x20656854	0x626d756e	0x20737265	0x20657261	0x66666964	0x6e657265	0x4e002e74	0x6968746f
0x10010020	0x6820676e	0x65707061	0x2e64656e	0x65685400	0x6d756e20	0x73726562	0x65726120	0x75716520
0x10010040	0x002e6c61	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010160	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010180	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100101a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100101c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

The Registers window on the right shows the register values:

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10000000
\$sp	29	0x7ffffefffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

The Mars Messages window shows the message: "Assemble: operation completed successfully."

Figure 17: MIPS Simulator after assembling the if-then-else code.

This screenshot is identical to Figure 17, showing the MIPS Simulator interface with the same assembly code, memory dump, and register values.

Figure 18: Mips simulator after running through the if loop and jumping immediately to the numbersEqual branch.

The screenshot shows the Mars MIPS simulator interface. The assembly code window displays the following instructions:

```

Text Segment
Bkpt Address Code Basic Source
0x00400004 0x20090014 addi $9,$0,0x00000014 9: addi $t1,$zero,20#can also change this number
0x00400008 0x11090009 beq $8,$9,0x00000009 10: if: beq $t0,$t1,numbersEqual#if the two numbers are equal it will print the numbers are equal
0x0040000c 0x15090002 bne $8,$9,0x00000002 12: else: bne $t0,$t1,numbersDifferent#else if the numbers are not equal it will print that the numbers are diff...
0x00400010 0x2402000a addiu $2,$0,0x0000000a 13: li $v0,10
0x00400014 0x2402000c syscall 14: syscall
0x00400018 0x24020004 addiu $2,$0,0x00000004 17: li $v0,4
0x0040001c 0x3c011001 lui $1,0x00001001 18: la $a0,message
0x00400020 0x34240000 ori $4,$1,0x00000000
0x00400024 0x0000000c syscall 19: syscall
0x00400028 0x2402000a addiu $2,$0,0x0000000a 20: li $v0,10
0x0040002c 0x0000000c syscall 21: syscall
0x00400030 0x24020004 addiu $2,$0,0x00000004 24: li $v0,4
0x00400034 0x3c011001 lui $1,0x00001001 25: la $a0,message3
0x00400038 0x34240002d ori $4,$1,0x0000002d
0x0040003c 0x0000000c syscall 26: syscall
0x00400040 0x2402000a addiu $2,$0,0x0000000a 27: li $v0,10
0x00400044 0x0000000c syscall 28: syscall

```

The Registers window shows the following register values:

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x10010000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x1001002d
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x000000014
\$t1	9	0x000000014
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7fffffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400048
hi		0x00000000
lo		0x00000000

The Data Segment window shows memory dump starting at address 0x10010000. The bottom status bar indicates "The numbers are equal." and "-- program is finished running --".

Figure 19: MIPS after finishing if-then-else statement.

After mips runs on the if then else statement it prints on the bottom that the numbers are equal as expected because 20 is equal to 20, it is correctly comparing the two registers where we have stored our values.

2.5 Calling myadd Function in MIPS

In this section we will be calling a function that adds two numbers, we can easily do that in practice but we are using a function to do it instead so that we can show how a function call works in MIPS.

```
GUTIERREZ_5_CALLING_MY_ADD_FUNCTION.asm
1 .data
2 a: .word 100
3 b: .word 200
4 c: .word 0
5
6 .text
7 lw $s0, a
8 lw $s1, b
9 lw $s2, c
10
11 main:
12 jal myadd #Calls My add function to add two of the words
13 sw $t0, -4($sp)
14 li $v0, 10
15 syscall
16
17 myadd:
18 # a = b + c
19 add $t0, $s0, $s1 # adds the two words
20 sw $t0, c
21 jr $ra #returns to the main function until reaching the end of the functions body.
```

Figure 20: Assembly code for calling myadd in MIPS.

In this code we can see that we are using 3 static variables, a, b and c, each stored in \$s0, \$s1 and \$s2 respectively. We are using the jump and link command jal to call the function myadd that will add the numbers \$s0 and \$s1 storing them in \$t0 then it stores that in the address where C was stored.

The screenshot shows the Mars SIMulator interface with three main panes:

- Text Segment:** Displays the assembly code from Figure 20. It includes columns for Bkpt, Address, Code, Basic, Source, and Comment. The code is color-coded to highlight labels, instructions, and comments.
- Data Segment:** Displays memory dump tables for Address, Value (+0), Value (+4), Value (+8), Value (+c), Value (+10), Value (+14), Value (+18), and Value (+1c). The first few rows show data for variables a, b, and c.
- Registers:** Shows the state of 32 registers (r0 to r31, \$zero to \$fp) with their names, numbers, and values.

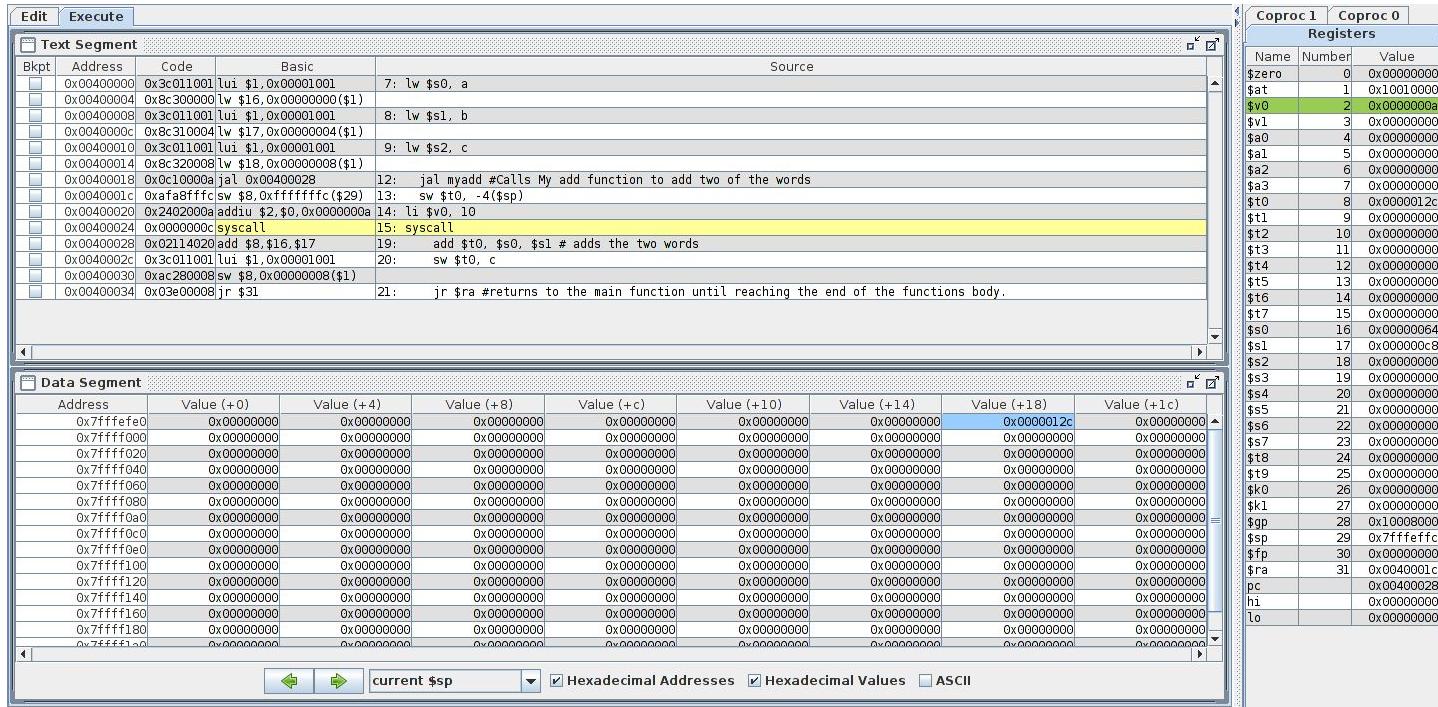
At the bottom, there are buttons for assembly, run, and halt, along with checkboxes for Hexadecimal Addresses, Hexadecimal Values, and ASCII. A message bar at the bottom indicates "Assemble: operation completed successfully."

Figure 21:MIPS simulator immediately after assembling the calling myadd function code.

The screenshot shows the MIPS simulator interface with the following details:

- Text Segment:** Displays assembly code for the `myadd` function. The code includes instructions for loading registers (\$s0, \$s1, \$s2), performing a jump to `myadd`, adding the values in `$s0` and `$s1` into `$t0`, and returning to the main function.
- Data Segment:** Shows memory starting at address `0x10010000` with all fields set to zero.
- Registers:** A column on the right lists registers from `$zero` to `$ra`, each initialized to zero.
- Toolbars and Buttons:** Includes standard simulation controls like step, run, and break, along with checkboxes for Hexadecimal Addresses, Hexadecimal Values, and ASCII.

Figure 22:MIPS simulator after loading the address of the instruction that will call myadd in order to perform the addition operation.

Figure 23: MIPS simulator after finishing the call of `myadd` function.

After the addition is performed the value is stored in the address where the value of C was previously stored which in this case is 0x7fffff8 as highlighted in blue in data segment.

2.6 Logical Operators in MIPS

In this section we will be using some of the MIPS logical operations.

```

GUTIERREZ_6_LOGICAL_OPERATOR.asm
1 # left shift
2 li $s0, 9
3 sll $t2, $s0, 4
4
5 # AND
6 li $t2, 0xdco
7 li $t1, 0x3c00
8 and $t0, $t1, $t2
9
10 # OR
11 or $t0, $t1, $t2
12
13 # NOR
14 li $t3, 0
15 nor $t0, $t1, $t3

```

Figure 24: Assembly code for MIPS operations.

We will be performing set less than, and, or, and nor on bits of information.

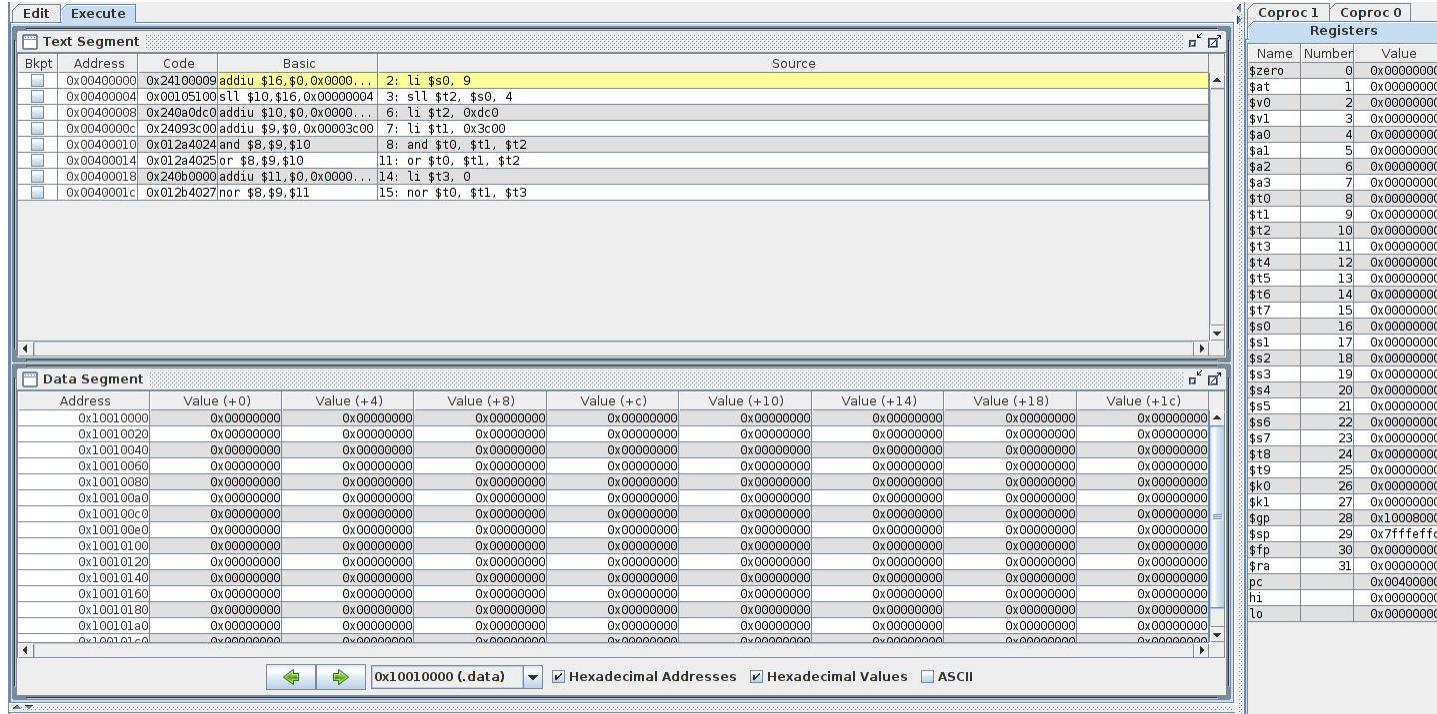


Figure 25: MIPS simulator immediately after assembling the logical operators code.

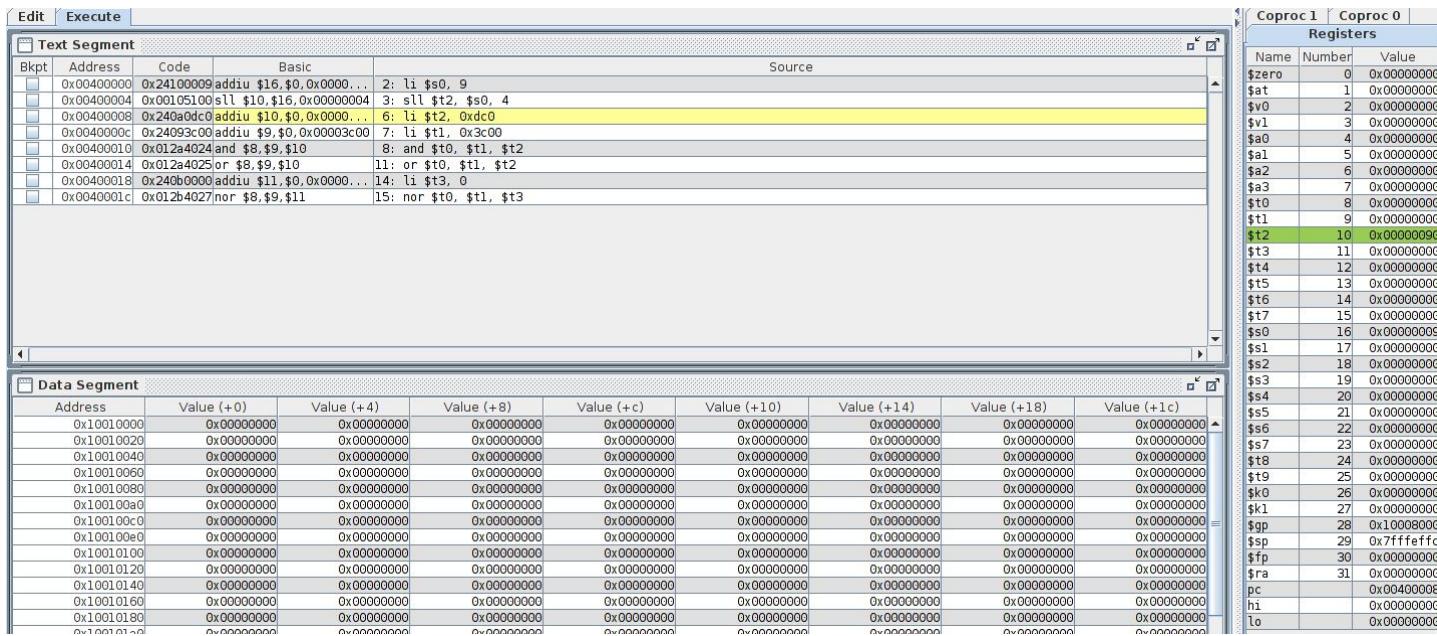


Figure 26: MIPS simulator after storing the value 9 in register \$t2.

The screenshot shows a MIPS simulator interface. On the left, the assembly code in the Text Segment window is as follows:

```

    .Text
    .Bkpt Address Code Basic Source
    0x00400000 0x24100009 addiu $16,$0,0x0000... 2: li $s0, 9
    0x00400004 0x00051000 sll $10,$16,0x00000004 3: sll $t2, $s0, 4
    0x00400008 0x240a0dc0 addiu $10,$0,0x0000... 6: li $t2, 0xdc0
    0x0040000c 0x24093c00 addiu $9,$0,0x00003c00 7: li $t1, 0x3c00
    0x00400010 0x012a4024 and $8,$9,$10 8: and $t0, $t1, $t2
    0x00400014 0x012a4025 or $8,$9,$10 11: or $t0, $t1, $t2
    0x00400018 0x240b0000 addiu $11,$0,0x0000... 14: li $t3, 0
    0x0040001c 0x012b4027 nor $8,$9,$11 15: nor $t0, $t1, $t3
  
```

The register window on the right shows the state after the replacement:

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000dc0
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000009
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7ffffeffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x0040000c
hi		0x00000000
lo		0x00000000

Figure 27: MIPS simulator after replacing the value stored in \$t2 to 0xdc0.

The screenshot shows a MIPS simulator interface. On the left, the assembly code in the Text Segment window is as follows:

```

    .Text
    .Bkpt Address Code Basic Source
    0x00400000 0x24100009 addiu $16,$0,0x0000... 2: li $s0, 9
    0x00400004 0x00051000 sll $10,$16,0x00000004 3: sll $t2, $s0, 4
    0x00400008 0x240a0dc0 addiu $10,$0,0x0000... 6: li $t2, 0xdc0
    0x0040000c 0x24093c00 addiu $9,$0,0x00003c00 7: li $t1, 0x3c00
    0x00400010 0x012a4024 and $8,$9,$10 8: and $t0, $t1, $t2
    0x00400014 0x012a4025 or $8,$9,$10 11: or $t0, $t1, $t2
    0x00400018 0x240b0000 addiu $11,$0,0x0000... 14: li $t3, 0
    0x0040001c 0x012b4027 nor $8,$9,$11 15: nor $t0, $t1, $t3
  
```

The register window on the right shows the state after the replacement:

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$t0	8	0x000003dc
\$t1	9	0x000003dc
\$t2	10	0x00000dc0
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000009
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7ffffeffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400018
hi		0x00000000
lo		0x00000000

Figure 28: MIPS simulator after replacing the value stored in \$t0 to 0xdc0.

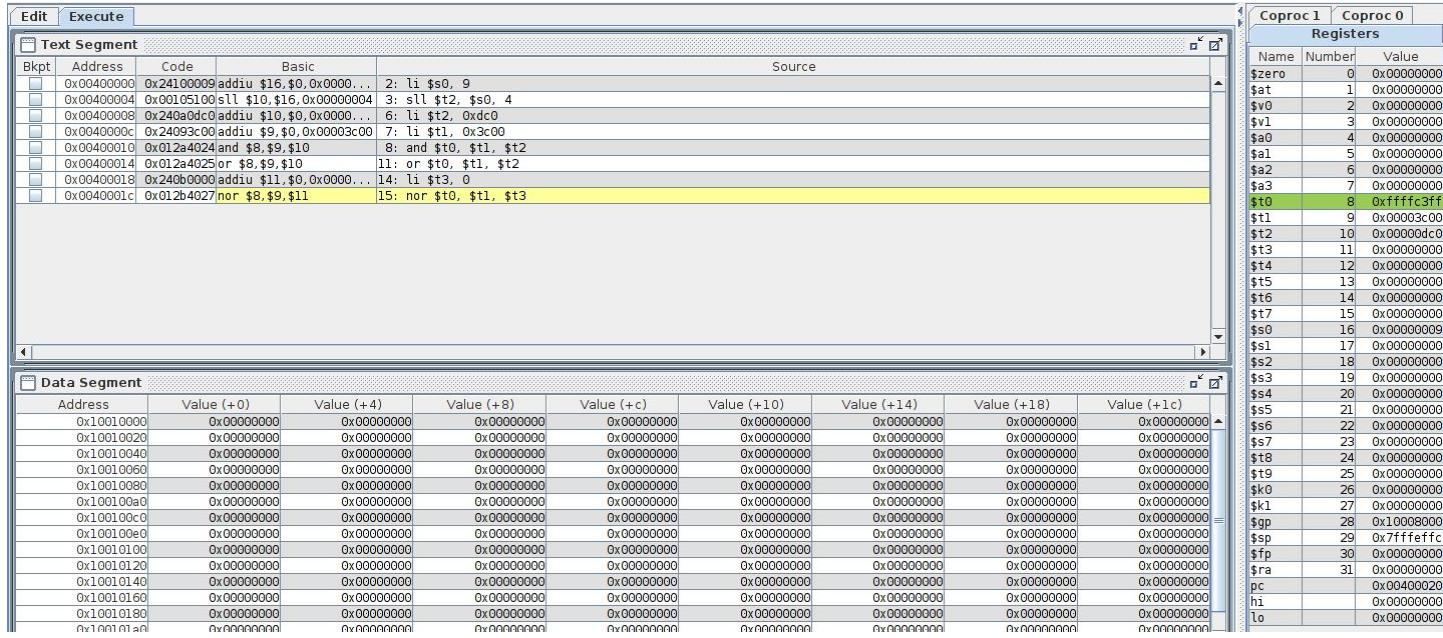


Figure 29:LOGICAL NOR on register \$t0.

2.7 Using arrays in MIPS

We will be testing arrays in MIPS now.

```
GUTIERREZ_7_USING_ARRAYS.asm
1 .data
2 h: .word 20
3 A: .word 0-400
4
5 .text
6 la $t1, A
7 lw $s2, h
8
9 #initializing A[300] to 13
10 li $t2, 13
11 sw $t2, 1200($t1)
12 lw $t0, 1200($t1)
13 add $t0, $s2, $t0
14 sw $t0, 1200($t1)
```

Figure 30: Assembly code for using an Array in MIPS.

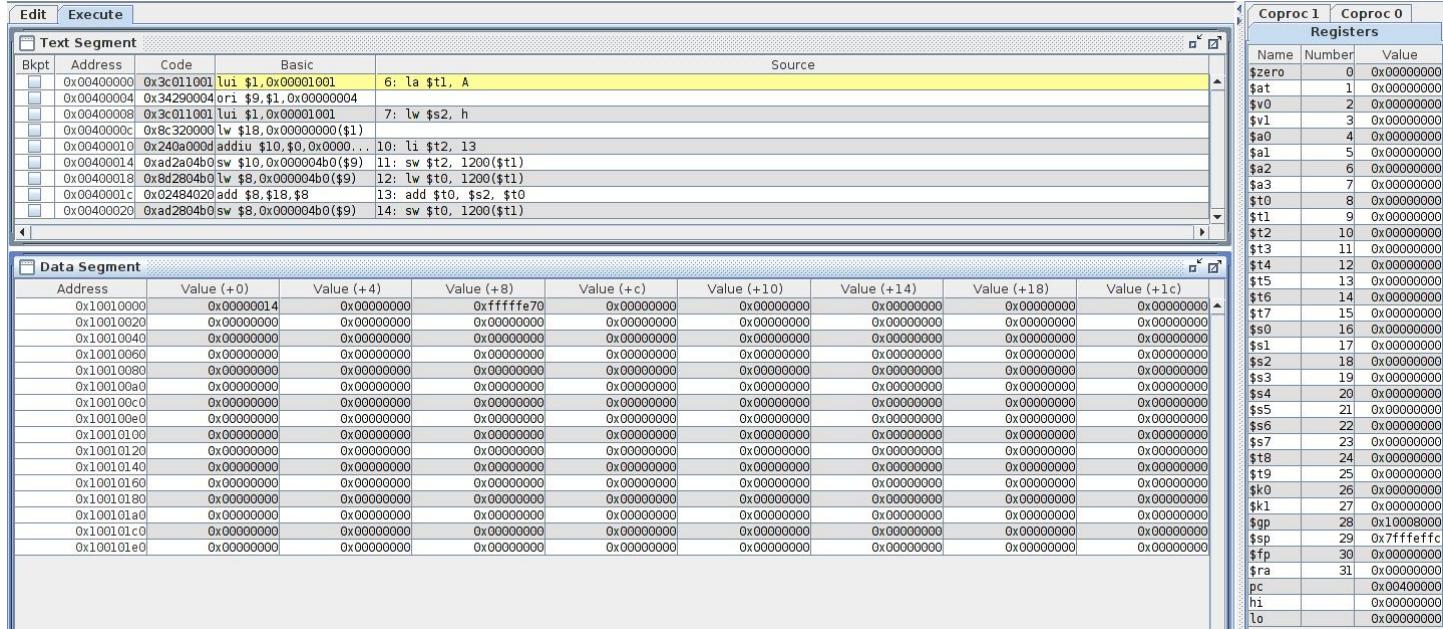


Figure 31:MIPS simulator immediately after assembling the code for using an array.

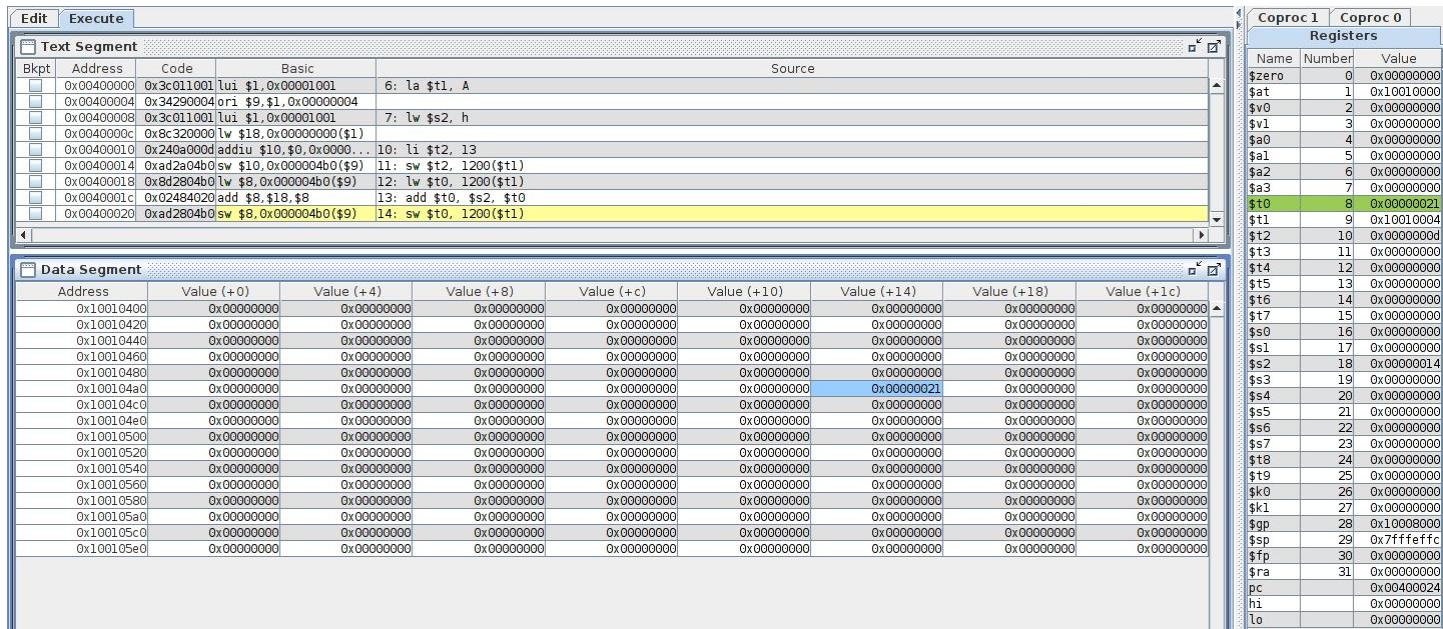


Figure 32:MIPS simulator after finishing accessing an array.

As we can see the value of 21 is stored in the array because it is set to 13 initially in hexadecimal format at the address A[300] which is highlighted in blue. But we are adding the value of h to the same location.

PART 2 LINUX 64 BIT COMPILER

The purpose of this section is to test the way that local variables, static variables, while loops, for loops, if-then-else statements, calling a function, logical operators, and using arrays is handled on linux using a 64 bit compiler using gcc to compile our code and GDB to debug it. We will be testing simple code for each of the 8 cases and we will be observing how the variables are handled as well as how data is stored and accessed. We will also be noting what is similar and different when compared to the other 3 platforms that will be observed during this exam. We will also note that Linux 64 bit compiler uses little Endian, that means it stores the least significant byte of information in memory into the lowest address.

3.1 Local variables in Linux 64 Bit Compiler

In this section we will be testing local variables in Linux 64 bit compiler.

```
GUTIERREZ_1_LOCAL_VARIABLES.c
1 //Using 5 Local Variables to test Local Variables in C
2 void main() {
3     int a = 1;
4     int b = 2;
5     int c = 3;
6     int d = 4;
7     int e = 5;
8     a = b + c;
9     d = a - e;
10 }
```

Figure 33: C code for initializing 5 local variables and adding 2 of them and subtracting 2 of them.

```
(gdb) x &a
0x7fffffffdd4c: 0x00000001
(gdb) x $rbp -8
0x7fffffffdd58: 0x00000000
(gdb) x &b
0x7fffffffdd50: 0x00000002
(gdb) x $rbp -0x4
0x7fffffffdd5c: 0x00000000
```

Figure 34: GDB return addresses for the variables a and b in memory.

As we can see the value of a is 1, b is 2 and the addresses are 64 bit. That is something that is new because it means that now since we are using a 64 bit compiler we are looking at larger addresses.

```
(gdb) next 1
9      d = a - e;
(gdb) next 1
10     }
(gdb) disassemble
Dump of assembler code for function main:
0x00000000004004d6 <+0>: push  %rbp
0x00000000004004d7 <+1>: mov   %rsp,%rbp
0x00000000004004da <+4>: movl $0x1,-0x14(%rbp)
0x00000000004004e1 <+11>: movl $0x2,-0x10(%rbp)
0x00000000004004e8 <+18>: movl $0x3,-0xc(%rbp)
0x00000000004004ef <+25>: movl $0x4,-0x8(%rbp)
0x00000000004004f6 <+32>: movl $0x5,-0x4(%rbp)
0x00000000004004fd <+39>: mov   -0x10(%rbp),%edx
0x0000000000400500 <+42>: mov   -0xc(%rbp),%eax
0x0000000000400503 <+45>: add   %edx,%eax
0x0000000000400505 <+47>: mov   %eax,-0x14(%rbp)
0x0000000000400508 <+50>: mov   -0x14(%rbp),%eax
0x000000000040050b <+53>: sub   -0x4(%rbp),%eax
0x000000000040050e <+56>: mov   %eax,-0x8(%rbp)
=> 0x0000000000400511 <+59>: nop
0x0000000000400512 <+60>: pop   %rbp
0x0000000000400513 <+61>: retq 
End of assembler dump.
```

Figure 35:Disassembly code for using local variables in Linux 64.

As different from MIPS instead of using \$sp we are using %rbp as the base pointer that is in 64 bits now and it's on linux as opposed to MIPS. the second command is to set the stack pointer equal to the base pointer which is the line mov %rsp,%rbp, after that the information is stored using an offset from the base pointer. The first value is stored using the command movl \$\$0x1, -0x14(%rbp) which is storing the value of 1 at an offset of -0x14 from the base pointer, the same

continues using different offsets for the remaining values. The last command to happen is to pop %rbp which frees the memory back for use.

3.2 Static variables in Linux 64 Bit Compiler

In this section we will be testing static variables on Linux 64 bit compiler.

```
GUTIERREZ_2_STATIC_VARIABLES.c
1 //Using 2 Globally Static Variables and using 3 Static Variables
2 static int a = 1;
3 static int b = 2;
4 void main() {
5
6     static int c = 3;
7     static int d = 4;
8     static int e = 5;
9     a = b + c;
10    d = a - e;
11 }
```

Figure 36: C code for using static variables in Linux 64 bit compiler.

```
Breakpoint 1, main () at GUTIERREZ_2_STATIC_VARIABLES.c:9
9     a = b + c;
(gdb) next 1
10    d = a - e;
(gdb) disassemble
Dump of assembler code for function main:
0x00000000004004d6 <+0>: push  %rbp
0x00000000004004d7 <+1>: mov   %rsp,%rbp
0x00000000004004da <+4>: mov   0x200b54(%rip),%edx      # 0x601034 <b>
0x00000000004004e0 <+10>: mov   0x200b52(%rip),%eax      # 0x601038 <c.1834>
0x00000000004004e6 <+16>: add   %edx,%eax
0x00000000004004e8 <+18>: mov   %eax,0x200b42(%rip)      # 0x601030 <a>
=> 0x00000000004004ee <+24>: mov   0x200b3c(%rip),%edx      # 0x601030 <a>
0x00000000004004f4 <+30>: mov   0x200b42(%rip),%eax      # 0x60103c <e.1836>
0x00000000004004fa <+36>: sub   %eax,%edx
0x00000000004004fc <+38>: mov   %edx,%eax
0x00000000004004fe <+40>: mov   %eax,0x200b3c(%rip)      # 0x601040 <d.1835>
0x0000000000400504 <+46>: nop
0x0000000000400505 <+47>: pop   %rbp
0x0000000000400506 <+48>: retq
End of assembler dump.
(gdb) ■
```

Figure 37:Disassembly information form GDB using static variables.

As we can see when we are storing static variables we have offsets that are very large and positive. That is because when we are using static variables in linux as opposed to MIPS instead of storing the values in a data segment it stores the values closer to an address to the instruction

pointer, and that is why the offset from the instruction pointer is so large. In this case instead of pc the instruction pointer is %rip.

```
(gdb) x &a
0x601030 <a>: 0x00000005
(gdb) x &b
0x601034 <b>: 0x00000002
(gdb) x &c
0x601038 <c.1834>: 0x00000003
(gdb) x &d
0x601040 <d.1835>: 0x00000004
(gdb) x &e
0x60103c <e.1836>: 0x00000005
```

Figure 38: GDB output for addresses of a, b, c, d and e as well as their values.

As we can see the values for the addresses are not in 64 bit like they were for the local variables that is because they are at a larger offset from the instruction pointer and are therefore smaller values.

```
(gdb) x/i $pc
=> 0x4004ee <main+24>: mov    0x200b3c(%rip),%edx      # 0x601030 <a>
```

Figure 39: Value of the next instruction in GDB debugger.

The offset is large from the \$rip which is why the address is smaller.

3.3 While Loop in Linux 64 Bit Compiler

In this section we will be testing while loops in 64 bit linux compiler.

```
GUTIERREZ_3_WHILE_LOOP.c
1 #include <stdio.h>
2 int main () {
3     int Local_Variable = 0;
4     while( Local_Variable < 10 ) {
5         printf("Local_Variable Value is : %d\n", Local_Variable);
6         Local_Variable++;
7     }
8
9     return 0;
10 }
```

Figure 40: C code for while loop for Linux 64 bit compiler.

```

0x000000000000400526 <+0>:    push   %rbp
0x000000000000400527 <+1>:    mov    %rsp,%rbp
0x00000000000040052a <+4>:    sub    $0x10,%rsp
0x00000000000040052e <+8>:    movl   $0x0,-0x4(%rbp)
0x000000000000400535 <+15>:   jmp    0x40054f <main+41>
0x000000000000400537 <+17>:   mov    -0x4(%rbp),%eax
=> 0x00000000000040053a <+20>:   mov    %eax,%esi
0x00000000000040053c <+22>:   mov    $0x4005e4,%edi
0x000000000000400541 <+27>:   mov    $0x0,%eax
0x000000000000400546 <+32>:   callq  0x400400 <printf@plt>
0x00000000000040054b <+37>:   addl   $0x1,-0x4(%rbp)
0x00000000000040054f <+41>:   cmpl   $0x9,-0x4(%rbp)
0x000000000000400553 <+45>:   jle    0x400537 <main+17>
0x000000000000400555 <+47>:   mov    $0x0,%eax
0x00000000000040055a <+52>:   leaveq 
0x00000000000040055b <+53>:   retq  
End of assembler dump.

```

Figure 41: GDB disassembly for while loop Linux 64 bits.

In this situation jmp is the jump command for the while loop that will be running the while loop to check the condition. Jle is the command for jump if less than which is what we are using in our while loop as we increment our iterator.

```

Local_Variable Value is : 1
6          Local_Variable++;
(gdb) nexti
4          while( Local_Variable < 10 ) {
(gdb) x $Local_Variable
Value can't be converted to integer.
(gdb) x &Local_Variable
0x7fffffffdd5c: 0x00000002

```

Figure 42: GDB address and value for Local_variale, in this case the value is 0x00000002 and the address is in 64 bits and is 0x7fffffffdd5c where the variable is stored, the while loop will continue until it is finished and that will be only when and if the value of Local_variable is no longer less than 10.

3.4 For Loop in Linux 64 Bit Compiler

In this section we will be testing for loop on the Linux 64 bit compiler.

```
GUTIERREZ_4_FOR_LOOP.c
1 #include <stdio.h>
2 int main () {
3     int Local_Variable;
4     for( Local_Variable = 10; Local_Variable < 20; Local_Variable = Local_Variable + 1 ){
5         printf("Local_Variable's Value is: %d\n", Local_Variable);
6     }
7     return 0;
8 }
```

Figure 43: C code for For loop in Linux 64 bits.

```
Dump of assembler code for function main:
0x00000000000400526 <+0>:    push   %rbp
0x00000000000400527 <+1>:    mov    %rsp,%rbp
0x0000000000040052a <+4>:    sub    $0x10,%rsp
0x0000000000040052e <+8>:    movl   $0xa,-0x4(%rbp)
0x00000000000400535 <+15>:   jmp    0x40054f <main+41>
0x00000000000400537 <+17>:   mov    -0x4(%rbp),%eax
0x0000000000040053a <+20>:   mov    %eax,%esi
0x0000000000040053c <+22>:   mov    $0x4005e8,%edi
0x00000000000400541 <+27>:   mov    $0x0,%eax
0x00000000000400546 <+32>:   callq  0x400400 <printf@plt>
=> 0x0000000000040054b <+37>:  addl   $0x1,-0x4(%rbp)
0x0000000000040054f <+41>:   cmpl   $0x13,-0x4(%rbp)
0x00000000000400553 <+45>:   jle    0x400537 <main+17>
0x00000000000400555 <+47>:   mov    $0x0,%eax
0x0000000000040055a <+52>:   leaveq 
0x0000000000040055b <+53>:   retq  
End of assembler dump.
```

Figure 44: GDB disassembly for for loop Linux 64 bits.

In this situation jmp is the jump command for the for loop that will be running the while loop to check the condition. Jle is the command for jump if less than which is what we are using in our for loop as we increment our iterator.

```
(gdb) next
Local_Variable's Value is: 16
4     for( Local_Variable = 10; Local_Variable < 20; Local_Variable = Local_Variable + 1 ){
(gdb) next
5         printf("Local_Variable's Value is: %d\n", Local_Variable);
(gdb) next
Local_Variable's Value is: 17
4     for( Local_Variable = 10; Local_Variable < 20; Local_Variable = Local_Variable + 1 ){
(gdb) next
5         printf("Local_Variable's Value is: %d\n", Local_Variable);
(gdb) next
Local_Variable's Value is: 18
4     for( Local_Variable = 10; Local_Variable < 20; Local_Variable = Local_Variable + 1 ){
(gdb) next
5         printf("Local_Variable's Value is: %d\n", Local_Variable);
(gdb) next
Local_Variable's Value is: 19
4     for( Local_Variable = 10; Local_Variable < 20; Local_Variable = Local_Variable + 1 ){
(gdb) next
7     return 0;
```

Figure 45:GDB output for running the next command a ve times to see the value of Local_variable and what happens after the for loop is finished.1

```
(gdb) disassemble
Dump of assembler code for function main:
0x000000000400526 <+0>:    push   %rbp
0x000000000400527 <+1>:    mov    %rsp,%rbp
0x00000000040052a <+4>:    sub    $0x10,%rsp
0x00000000040052e <+8>:    movl   $0xa,-0x4(%rbp)
0x000000000400535 <+15>:   jmp    0x40054f <main+41>
0x000000000400537 <+17>:   mov    -0x4(%rbp),%eax
0x00000000040053a <+20>:   mov    %eax,%esi
0x00000000040053c <+22>:   mov    $0x4005e8,%edi
0x000000000400541 <+27>:   mov    $0x0,%eax
0x000000000400546 <+32>:   callq  0x400400 <printf@plt>
0x00000000040054b <+37>:   addl   $0x1,-0x4(%rbp)
0x00000000040054f <+41>:   cmpl   $0x13,-0x4(%rbp)
0x000000000400553 <+45>:   jle    0x400537 <main+17>
=> 0x000000000400555 <+47>:   mov    $0x0,%eax
0x00000000040055a <+52>:   leaveq 
0x00000000040055b <+53>:   retq  
End of assembler dump.
```

Figure 44:Disassemble for for loop in linux 64 bits.

3.5 If-then-else statements in Linux 64 Bit Compiler

In this section we will be testing if-then-else statements in Linux 64 bit compiler, we will be checking when a condition is or isn't met and how branching works in linux.

```
GUTIERREZ_5_IF_THEN_ELSE_LOOPS.c x
1 #include <stdio.h>
2 int main () {
3     int Local_Variable = 5;
4     if( Local_Variable < 20 ) {
5         printf("Local_Variable is less than 20\n" );
6     }
7     else {
8         printf("Local_Variable is not less than 20\n" );
9     }
10    printf("Local_Variable's value is : %d\n", Local_Variable);
11    return 0;
12 }
```

Figure 47: C code for if-then-else case for linux 64 bit.

```
Dump of assembler code for function main:
0x000000000000400566 <+0>:    push   %rbp
0x000000000000400567 <+1>:    mov    %rsp,%rbp
0x00000000000040056a <+4>:    sub    $0x10,%rsp
=> 0x00000000000040056e <+8>:    movl   $0x5,-0x4(%rbp)
0x000000000000400575 <+15>:   cmpl   $0x13,-0x4(%rbp)
0x000000000000400579 <+19>:   jg     0x400587 <main+33>
0x00000000000040057b <+21>:   mov    $0x400638,%edi
0x000000000000400580 <+26>:   callq  0x400430 <puts@plt>
0x000000000000400585 <+31>:   jmp    0x400591 <main+43>
0x000000000000400587 <+33>:   mov    $0x400658,%edi
0x00000000000040058c <+38>:   callq  0x400430 <puts@plt>
0x000000000000400591 <+43>:   mov    -0x4(%rbp),%eax
0x000000000000400594 <+46>:   mov    %eax,%esi
0x000000000000400596 <+48>:   mov    $0x400680,%edi
0x00000000000040059b <+53>:   mov    $0x0,%eax
0x0000000000004005a0 <+58>:   callq  0x400440 <printf@plt>
0x0000000000004005a5 <+63>:   mov    $0x0,%eax
0x0000000000004005aa <+68>:   leaveq 
0x0000000000004005ab <+69>:   retq 

End of assembler dump.
```

Figure 48: Disassembly for GDB if-then-else statement in Linux 64 bits.

```
Starting program: /home/jgutierrez/Desktop/GUTIERREZ_5_IF_THEN_ELSE_LOOPS.c:3
Breakpoint 1, main () at GUTIERREZ_5_IF_THEN_ELSE_LOOPS.c:3
3      int Local_Variable = 5;
(gdb) next
4      if( Local_Variable < 20 ) {
(gdb) next
5      printf("Local_Variable is less than 20\n" );
```

Figure 49: GDB output reading the if statement after the value of Local_variable is set to 5.

```
(gdb) disassemble
Dump of assembler code for function main:
0x000000000000400566 <+0>:    push   %rbp
0x000000000000400567 <+1>:    mov    %rsp,%rbp
0x00000000000040056a <+4>:    sub    $0x10,%rsp
0x00000000000040056e <+8>:    movl   $0x5,-0x4(%rbp)
0x000000000000400575 <+15>:   cmpl   $0x13,-0x4(%rbp)
0x000000000000400579 <+19>:   jg     0x400587 <main+33>
0x00000000000040057b <+21>:   mov    $0x400638,%edi
0x000000000000400580 <+26>:   callq  0x400430 <puts@plt>
0x000000000000400585 <+31>:   jmp    0x400591 <main+43>
0x000000000000400587 <+33>:   mov    $0x400658,%edi
0x00000000000040058c <+38>:   callq  0x400430 <puts@plt>
=> 0x000000000000400591 <+43>:   mov    -0x4(%rbp),%eax
0x000000000000400594 <+46>:   mov    %eax,%esi
0x000000000000400596 <+48>:   mov    $0x400680,%edi
0x00000000000040059b <+53>:   mov    $0x0,%eax
0x0000000000004005a0 <+58>:   callq  0x400440 <printf@plt>
0x0000000000004005a5 <+63>:   mov    $0x0,%eax
0x0000000000004005aa <+68>:   leaveq 
0x0000000000004005ab <+69>:   retq 

End of assembler dump.
```

Figure 50: GDB disassembly window for if then else statement calling the print statement when the condition required is met.

```
(gdb) disassemble
Dump of assembler code for function main:
0x000000000400566 <+0>:    push   %rbp
0x000000000400567 <+1>:    mov    %rsp,%rbp
0x00000000040056a <+4>:    sub    $0x10,%rsp
0x00000000040056e <+8>:    movl   $0x5,-0x4(%rbp)
0x000000000400575 <+15>:   cmpl   $0x13,-0x4(%rbp)
0x000000000400579 <+19>:   jg    0x400587 <main+33>
0x00000000040057b <+21>:   mov    $0x400638,%edi
0x000000000400580 <+26>:   callq  0x400430 <puts@plt>
0x000000000400585 <+31>:   jmp    0x400591 <main+43>
0x000000000400587 <+33>:   mov    $0x400658,%edi
0x00000000040058c <+38>:   callq  0x400430 <puts@plt>
0x000000000400591 <+43>:   mov    -0x4(%rbp),%eax
0x000000000400594 <+46>:   mov    %eax,%esi
0x000000000400596 <+48>:   mov    $0x400680,%edi
0x00000000040059b <+53>:   mov    $0x0,%eax
0x0000000004005a0 <+58>:   callq  0x400440 <printf@plt>
=> 0x0000000004005a5 <+63>:   mov    $0x0,%eax
0x0000000004005aa <+68>:   leaveq 
0x0000000004005ab <+69>:   retq  
End of assembler dump.
```

Figure 51: Disassembly GDB for if-then else statement at the end of the program, again it is deallocating memory at the end of the function calls just like MIPS does.

3.6 Calling Myadd Function in Linux 64 Bit Compiler

In this section we will be calling a function called MyAdd in the Linux 64 bit compiler.

```
GUTIERREZ_6_CALLING_MY_ADD.c  *
1 int myadd (int& const Word1, int& const Word2) {
2     return Word1 + Word2;
3 }
4 int main() {
5     int Word1 = 0xffffffff;
6     int Word2 = 0xffffffff;
7     int Word3 = 0;
8     Word3 = myadd(Word1, Word2);
9     return 0;
10 }
```

Figure 52: C code for calling Myadd function for linux 64 bit compiler.

```
(gdb) disassemble
Dump of assembler code for function main:
0x0000000000004004ea <+0>:    push    %rbp
0x0000000000004004eb <+1>:    mov     %rsp,%rbp
0x0000000000004004ee <+4>:    sub    $0x10,%rsp
=> 0x0000000000004004f2 <+8>:    movl   $0x7fffffff,-0xc(%rbp)
0x0000000000004004f9 <+15>:   movl   $0xffffffff,-0x8(%rbp)
0x000000000000400500 <+22>:   movl   $0x0,-0x4(%rbp)
0x000000000000400507 <+29>:   mov    -0x8(%rbp),%edx
0x00000000000040050a <+32>:   mov    -0xc(%rbp),%eax
0x00000000000040050d <+35>:   mov    %edx,%esi
0x00000000000040050f <+37>:   mov    %eax,%edi
0x000000000000400511 <+39>:   callq  0x4004d6 <myadd>
0x000000000000400516 <+44>:   mov    %eax,-0x4(%rbp)
0x000000000000400519 <+47>:   mov    $0x0,%eax
0x00000000000040051e <+52>:   leaveq 
0x00000000000040051f <+53>:   retq 
End of assembler dump.
```

Figure 53: Disassembly code for myadd function using 2 variables stored in lines +8 and +15.

```
Breakpoint 1, main () at GUTIERREZ_6_CALLING_MY_ADD.c:5
5      int Word1 = 0x7fffffff;
(gdb) next
6      int Word2 = 0xffffffff;
(gdb) next
7      int Word3 = 0;
```

Figure 54: Output for running next 3 times after running the program in disassembly windows for GDB.

```
(gdb) next
8      Word3 = myadd(Word1, Word2);
(gdb) x &Word3
0x7fffffffdd5c: 0x00000000
(gdb) next
9      return 0;
(gdb) x &Word3
0x7fffffffdd5c: 0x7fffffe
```

Figure 55: GDB output value for the value stored in word3 which is the largest possible positive number -1 which turns out to be 0x7fffffe which is correct in this case. The function was called properly to add the two values.

```
(gdb) disassemble
Dump of assembler code for function main:
0x00000000004004ea <+0>:    push   %rbp
0x00000000004004eb <+1>:    mov    %rsp,%rbp
0x00000000004004ee <+4>:    sub    $0x10,%rsp
0x00000000004004f2 <+8>:    movl   $0x7fffffff,-0xc(%rbp)
0x00000000004004f9 <+15>:   movl   $0xffffffff,-0x8(%rbp)
0x0000000000400500 <+22>:   movl   $0x0,-0x4(%rbp)
0x0000000000400507 <+29>:   mov    -0x8(%rbp),%edx
0x000000000040050a <+32>:   mov    -0xc(%rbp),%eax
0x000000000040050d <+35>:   mov    %edx,%esi
0x000000000040050f <+37>:   mov    %eax,%edi
0x0000000000400511 <+39>:   callq  0x4004d6 <myadd>
0x0000000000400516 <+44>:   mov    %eax,-0x4(%rbp)
0x0000000000400519 <+47>:   mov    $0x0,%eax
=> 0x000000000040051e <+52>: leaveq 
0x000000000040051f <+53>: retq 
End of assembler dump.
```

Figure 56: GDB disassembly for calling myadd function after the program has finished running, again now it is deallocating memory.

3.7 Logical Operators in Linux 64 Bit Compiler

In this case we will be running logical operations on linux 64 bit compiler.

```
GUTIERREZ_7_LOGICAL_OPERATOR.c
1 void main() {
2     static int s0 = 9;
3     static int t1 = 0x3c00;
4     static int t2 = 0xdc0;
5     static int t3 = 0;
6
7     t3 = s0 << 4;
8     static int t0 = 0;
9
10    t0 = t1 & t2;
11
12    t0 = t1 | t2;
13
14    t0 = ~t1;|
15 }
```

Figure 57: C code for logical operators that will be run in Linux 64 bit.

```
(gdb) disassemble
Dump of assembler code for function main:
0x00000000004004d6 <+0>: push %rbp
0x00000000004004d7 <+1>: mov %rsp,%rbp
0x00000000004004da <+4>: mov 0x200b50(%rip),%eax      # 0x601030 <s0.1832>
0x00000000004004e0 <+10>: shl $0x4,%eax
0x00000000004004e3 <+13>: mov %eax,0x200b57(%rip)      # 0x601040 <t3.1835>
=> 0x00000000004004e9 <+19>: mov 0x200b45(%rip),%edx      # 0x601034 <t1.1833>
0x00000000004004ef <+25>: mov 0x200b43(%rip),%eax      # 0x601038 <t2.1834>
0x00000000004004f5 <+31>: and %edx,%eax
0x00000000004004f7 <+33>: mov %eax,0x200b47(%rip)      # 0x601044 <t0.1836>
0x00000000004004fd <+39>: mov 0x200b31(%rip),%edx      # 0x601034 <t1.1833>
0x0000000000400503 <+45>: mov 0x200b2f(%rip),%eax      # 0x601038 <t2.1834>
0x0000000000400509 <+51>: or %edx,%eax
0x000000000040050b <+53>: mov %eax,0x200b33(%rip)      # 0x601044 <t0.1836>
0x0000000000400511 <+59>: mov 0x200b1d(%rip),%eax      # 0x601034 <t1.1833>
0x0000000000400517 <+65>: not %eax
0x0000000000400519 <+67>: mov %eax,0x200b25(%rip)      # 0x601044 <t0.1836>
0x000000000040051f <+73>: nop
0x0000000000400520 <+74>: pop %rbp
0x0000000000400521 <+75>: retq
End of assembler dump.
```

Figure 58: GDB disassembly for logical operations. After compiling the code.

```
Breakpoint 1, main () at GUTIERREZ_7_LOGICAL_OPERATOR.c:7
7      t3 = s0 << 4;
(gdb) x &t3
0x601040 <t3.1835>:    0x00000000
(gdb) next 1
10     t0 = t1 & t2;
(gdb) x &t0
0x601044 <t0.1836>:    0x00000000
(gdb) next 1
12     t0 = t1 | t2;
(gdb) x &t0
0x601044 <t0.1836>:    0x00000c00
(gdb) next 1
14     t0 = ~t1;
(gdb) x &t0
0x601044 <t0.1836>:    0x00003dc0
(gdb) next 1
15   }
(gdb) x &t0
0x601044 <t0.1836>:    0xfffffc3ff
```

Figure 59: Output for the values at each instance of the logical operations.

As we can see the bitwise and, or not are all interpreted similarly to MIPS.

```
12     t0 = t1 | t2;
(gdb) x &t0
0x601044 <t0.1836>:    0x00000c00
```

Figure 60: GDB output and value of the variable t0 after performing the logical operation or between t1 and t2.

```
(gdb) disassembled
Dump of assembler code for function main:
0x00000000004004d6 <+0>:    push   %rbp
0x00000000004004d7 <+1>:    mov    %rsp,%rbp
0x00000000004004da <+4>:    mov    0x200b50(%rip),%eax      # 0x601030 <s0.1832>
0x00000000004004e0 <+10>:   shl    $0x4,%eax
0x00000000004004e3 <+13>:   mov    %eax,0x200b57(%rip)    # 0x601040 <t3.1833>
0x00000000004004e9 <+19>:   mov    0x200b45(%rip),%edx      # 0x601034 <t1.1833>
0x00000000004004ef <+25>:   mov    0x200b43(%rip),%eax      # 0x601038 <t2.1834>
0x00000000004004f5 <+31>:   and    %edx,%eax
0x00000000004004f7 <+33>:   mov    %eax,0x200b47(%rip)    # 0x601044 <t0.1836>
0x00000000004004fd <+39>:   mov    0x200b31(%rip),%edx      # 0x601034 <t1.1833>
0x0000000000400503 <+45>:   mov    0x200b2f(%rip),%eax      # 0x601038 <t2.1834>
0x0000000000400509 <+51>:   or     %edx,%eax
0x000000000040050b <+53>:   mov    %eax,0x200b33(%rip)    # 0x601044 <t0.1836>
0x0000000000400511 <+59>:   mov    0x200b1d(%rip),%eax      # 0x601034 <t1.1833>
0x0000000000400517 <+65>:   not    %eax
0x0000000000400519 <+67>:   mov    %eax,0x200b25(%rip)    # 0x601044 <t0.1836>
=> 0x000000000040051f <+73>:  nop
0x0000000000400520 <+74>:  pop    %rbp
0x0000000000400521 <+75>:  retq
End of assembler dump.
```

Figure 61: GDB disassembly window after terminating the logical operator program, again we can see that it is deallocating memory.

3.8 Using arrays in Linux 64 Bit Compiler

In this section we will be using arrays using linux 64 bit compiler, we will observe what happens in memory and how the program runs properly.

```
GUTIERREZ_8_USING_ARRAY.c  x
1 void main() {
2     static int h = 25;
3     static int A[100];
4     A[8] = 200;
5     A[12] = h + A[8];
6 }
```

Figure 62: C code for using arrays in Linux 64 bit compiler.

```
Dump of assembler code for function main:
0x00000000004004d6 <+0>:    push   %rbp
0x00000000004004d7 <+1>:    mov    %rsp,%rbp
=> 0x00000000004004da <+4>:   movl   $0xc8,0x200b9c(%rip)      # 0x601080 <A.1833+32>
0x00000000004004e4 <+14>:   mov    0x200b96(%rip),%edx      # 0x601080 <A.1833+32>
0x00000000004004ea <+20>:   mov    0x200b40(%rip),%eax      # 0x601030 <h.1832>
0x00000000004004f0 <+26>:   add    %edx,%eax
0x00000000004004f2 <+28>:   mov    %eax,0x200b98(%rip)    # 0x601090 <A.1833+48>
0x00000000004004f8 <+34>:   nop
0x00000000004004f9 <+35>:   pop    %rbp
0x00000000004004fa <+36>:   retq
End of assembler dump.
```

Figure 63: GDB disassembly window after compiling the C code for using arrays in Linux 64 bit compiler.

```
(gdb) disassemble
Dump of assembler code for function main:
0x00000000004004d6 <+0>:    push   %rbp
0x00000000004004d7 <+1>:    mov    %rsp,%rbp
0x00000000004004da <+4>:    movl   $0xc8,0x200b9c(%rip)      # 0x601080 <A.1833+32>
=> 0x00000000004004e4 <+14>:   mov    0x200b96(%rip),%edx      # 0x601080 <A.1833+32>
0x00000000004004ea <+20>:   mov    0x200b40(%rip),%eax      # 0x601030 <h.1832>
0x00000000004004f0 <+26>:   add    %edx,%eax
0x00000000004004f2 <+28>:   mov    %eax,0x200b98(%rip)      # 0x601090 <A.1833+48>
0x00000000004004f8 <+34>:   nop
0x00000000004004f9 <+35>:   pop    %rbp
0x00000000004004fa <+36>:   retq
End of assembler dump.
(gdb) x &h
0x601030 <h.1832>:    0x00000001
```

Figure 64: GDB value of H. The offset from the base pointer is where the value of h will be added in the array of 100 elements and it will be accessed even though other parts of the array are not even going to be initialized by default. That does not cause a problem because it is allowed, it is allowed because it is an accessible address.

```
(gdb) disassemble
Dump of assembler code for function main:
0x00000000004004d6 <+0>:    push   %rbp
0x00000000004004d7 <+1>:    mov    %rsp,%rbp
0x00000000004004da <+4>:    movl   $0xc8,0x200b9c(%rip)      # 0x601080 <A.1833+32>
0x00000000004004e4 <+14>:   mov    0x200b96(%rip),%edx      # 0x601080 <A.1833+32>
0x00000000004004ea <+20>:   mov    0x200b40(%rip),%eax      # 0x601030 <h.1832>
0x00000000004004f0 <+26>:   add    %edx,%eax
0x00000000004004f2 <+28>:   mov    %eax,0x200b98(%rip)      # 0x601090 <A.1833+48>
=> 0x00000000004004f8 <+34>:   nop
0x00000000004004f9 <+35>:   pop    %rbp
0x00000000004004fa <+36>:   retq
End of assembler dump.
```

Figure 65: GDB exiting after using the array again deallocated memory.

PART 3 LINUX 32 BIT ON RASPBERRY PI

The purpose of this section is to test the way that local variables, static variables, while loops, for loops, if-then-else statements, calling a function, logical operators, and using arrays is handled on Linux in 32 bits using a Raspberry Pi using gcc and gdb for compiling our code and for debugging it. We will be testing simple code for each of the 8 cases and we will be observing how the variables are handled as well as how data is stored and accessed. We will also be noting what is similar and different when compared to the other 3 platforms that will be observed during

this exam. We will also note that Raspberry Pi uses little Endian, that means it stores the least significant byte of information in memory into the lowest address.

4.1 Local variables on Raspberry Pi

In this section we will be testing local variables in Raspberry Pi.

```
GUTIERREZ_1_LOCAL_VARIABLES.c
1 //Using 5 Local Variables to test Local Variables in C
2 void main() {
3     int a = 1;
4     int b = 2;
5     int c = 3;
6     int d = 4;
7     int e = 5;
8     a = b + c;
9     d = a - e;
10 }
```

Figure 67:C code for initializing 5 local variables and adding 2 of them and subtracting 2 of them on Raspberry Pi.

```
(gdb) disassemble
Dump of assembler code for function main:
0x000103e8 <+0>:    push    {r11}          ; (str r11, [sp, #-4]!)
0x000103ec <+4>:    add    r11, sp, #0      ; (str r11, [sp, #-4]!)
0x000103f0 <+8>:    sub    sp, sp, #28
=> 0x000103f4 <+12>:   mov    r3, #1
0x000103f8 <+16>:   str    r3, [r11, #-8]
0x000103fc <+20>:   mov    r3, #2
0x00010400 <+24>:   str    r3, [r11, #-12]
0x00010404 <+28>:   mov    r3, #3
0x00010408 <+32>:   str    r3, [r11, #-16]
0x0001040c <+36>:   mov    r3, #4
0x00010410 <+40>:   str    r3, [r11, #-20]
0x00010414 <+44>:   mov    r3, #5
0x00010418 <+48>:   str    r3, [r11, #-24]
0x0001041c <+52>:   ldr    r2, [r11, #-12]
0x00010420 <+56>:   ldr    r3, [r11, #-16]
0x00010424 <+60>:   add    r3, r2, r3
0x00010428 <+64>:   str    r3, [r11, #-8]
0x0001042c <+68>:   ldr    r2, [r11, #-8]
0x00010430 <+72>:   ldr    r3, [r11, #-24]
0x00010434 <+76>:   rsb    r3, r3, r2
0x00010438 <+80>:   str    r3, [r11, #-20]
0x0001043c <+84>:   sub    sp, r11, #0
---Type <return> to continue, or q <return> to quit---
0x00010440 <+88>:   pop    {r11}          ; (ldr r11, [sp], #4)
0x00010444 <+92>:   bx    lr
```

Figure 68: GDB disassembly for using local variables in Raspberry Pi.

As we can see a major difference between MIPS, Linux 64 bit and cortex on the raspberry pi is the names of the instructions that are used. Another difference is the name of their registers. In

this case they are r11 for the base pointer and sp for the stack pointer. There are also more instructions here than in any of the previous 2 compilers and that is because the operations take more steps to compute than they do on MIPS or linux 64 bits.

```
(gdb) disassemble
Dump of assembler code for function main:
0x000103e8 <+0>:    push    {r11}          ; (str r11, [sp, #-4]!)
0x000103ec <+4>:    add     r11, sp, #0
0x000103f0 <+8>:    sub     sp, sp, #28
0x000103f4 <+12>:   mov     r3, #1
0x000103f8 <+16>:   str     r3, [r11, #-8]
0x000103fc <+20>:   mov     r3, #2
0x00010400 <+24>:   str     r3, [r11, #-12]
0x00010404 <+28>:   mov     r3, #3
0x00010408 <+32>:   str     r3, [r11, #-16]
0x0001040c <+36>:   mov     r3, #4
0x00010410 <+40>:   str     r3, [r11, #-20]
0x00010414 <+44>:   mov     r3, #5
0x00010418 <+48>:   str     r3, [r11, #-24]
0x0001041c <+52>:   ldr     r2, [r11, #-12]
0x00010420 <+56>:   ldr     r3, [r11, #-16]
0x00010424 <+60>:   add     r3, r2, r3
0x00010428 <+64>:   str     r3, [r11, #-8]
=> 0x0001042c <+68>: ldr     r2, [r11, #-8]
0x00010430 <+72>:   ldr     r3, [r11, #-24]
0x00010434 <+76>:   rsb     r3, r3, r2
0x00010438 <+80>:   str     r3, [r11, #-20]
0x0001043c <+84>:   sub     sp, r11, #0
---Type <return> to continue, or q <return> to quit---
0x00010440 <+88>:   pop    {r11}          ; (ldr r11, [sp], #4)
0x00010444 <+92>:   bx     lr
End of assembler dump.
```

Figure 68: GDB disassemble after running a few lines into storing local variables into the stack

using offsets to the stack pointer.

```
(gdb) x $r11 -8
0x7effef94: 0x00000001
(gdb) x &a
0x7effef94: 0x00000001
(gdb) x &b
0x7effef90: 0x00000002
(gdb) x &c
0x7effef8c: 0x00000003
(gdb) x &d
0x7effef88: 0x00000000
(gdb) x $r11 -0x4
0x7effef98: 0x00000000
(gdb) █
```

Figure 69: GDB addresses for variables a,b,c, and d for Raspberry Pi.

As we can see the addresses in this case are 32 bits instead of 64 bits as expected. The values are also stored from an offset to the stack pointer which is true in Linux 64 bits too and in MIPS but it has a different name here r11.

4.2 Static variables on Raspberry Pi

```
GUTIERREZ_2_STATIC_VARIABLES.c
1 //Using 2 Globally Static Variables and using 3 Static Variables
2 static int a = 1;
3 static int b = 2;
4 void main() {
5
6     static int c = 3;
7     static int d = 4;
8     static int e = 5;
9     a = b + c;
10    d = a - e;
11 }
```

Figure 70: C code for using static variables in Raspberry Pi.

```
(gdb) disassemble
Dump of assembler code for function main:
0x000103e8 <+0>: push   {r11}          ; (str r11, [sp, #-4]!)
0x000103ec <+4>: add    r11, sp, #0
0x000103f0 <+8>: ldr    r3, [pc, #60]   ; 0x10434 <main+76>
0x000103f4 <+12>: ldr    r2, [r3]
0x000103f8 <+16>: ldr    r3, [pc, #56]   ; 0x10438 <main+80>
0x000103fc <+20>: ldr    r3, [r3]
0x00010400 <+24>: add    r3, r2, r3
0x00010404 <+28>: ldr    r2, [pc, #48]   ; 0x1043c <main+84>
0x00010408 <+32>: str    r3, [r2]
=> 0x0001040c <+36>: ldr    r3, [pc, #40]   ; 0x1043c <main+84>
0x00010410 <+40>: ldr    r2, [r3]
0x00010414 <+44>: ldr    r3, [pc, #36]   ; 0x10440 <main+88>
0x00010418 <+48>: ldr    r3, [r3]
0x0001041c <+52>: rsb    r3, r3, r2
0x00010420 <+56>: ldr    r2, [pc, #28]   ; 0x10444 <main+92>
0x00010424 <+60>: str    r3, [r2]
0x00010428 <+64>: sub    sp, r11, #0
0x0001042c <+68>: pop    {r11}          ; (ldr r11, [sp], #4)
0x00010430 <+72>: bx    lr
0x00010434 <+76>: andeq r0, r2, r4, ror #11
0x00010438 <+80>: andeq r0, r2, r8, ror #11
0x0001043c <+84>: andeq r0, r2, r0, ror #11
0x00010440 <+88>: andeq r0, r2, r12, ror #11
0x00010444 <+92>: strdeq r0, [r2], -r0  ; <UNPREDICTABLE>
End of assembler dump.
```

Figure 71: Disassembly information from GDB using static variables.

As we can see when we are storing static variables we have offsets that are very large and positive. That is because when we are using static variables in linux as opposed to MIPS instead of storing the values in a data segment it stores the values closer to an address to the instruction pointer, and that is why the offset from the instruction pointer is so large. In this case instead of pc the instruction pointer is r11. Even though we are using a 32 bit compiler in this case the offset is still much larger than it would have been in MIPS.

```
(gdb) disassemble
Dump of assembler code for function main:
0x000103e8 <+0>:    push   {r11}          ; (str r11, [sp, #-4]!)
0x000103ec <+4>:    add    r11, sp, #0
0x000103f0 <+8>:    ldr    r3, [pc, #60]   ; 0x10434 <main+76>
0x000103f4 <+12>:   ldr    r2, [r3]
0x000103f8 <+16>:   ldr    r3, [pc, #56]   ; 0x10438 <main+80>
0x000103fc <+20>:   ldr    r3, [r3]
0x00010400 <+24>:   add    r3, r2, r3
0x00010404 <+28>:   ldr    r2, [pc, #48]   ; 0x1043c <main+84>
0x00010408 <+32>:   str    r3, [r2]
0x0001040c <+36>:   ldr    r3, [pc, #40]   ; 0x1043c <main+84>
0x00010410 <+40>:   ldr    r2, [r3]
0x00010414 <+44>:   ldr    r3, [pc, #36]   ; 0x10440 <main+88>
0x00010418 <+48>:   ldr    r3, [r3]
0x0001041c <+52>:   rsb    r3, r3, r2
0x00010420 <+56>:   ldr    r2, [pc, #28]   ; 0x10444 <main+92>
0x00010424 <+60>:   str    r3, [r2]
=> 0x00010428 <+64>: sub    sp, r11, #0
0x0001042c <+68>:   pop    {r11}          ; (ldr r11, [sp], #4)
0x00010430 <+72>:   bx    lr
0x00010434 <+76>:   andeq r0, r2, r4, ror #11
0x00010438 <+80>:   andeq r0, r2, r8, ror #11
0x0001043c <+84>:   andeq r0, r2, r0, ror #11
0x00010440 <+88>:   andeq r0, r2, r12, ror #11
0x00010444 <+92>:   strdeq r0, [r2], -r0   ; <UNPREDICTABLE>
End of assembler dump.
(gdb)
```

Figure 72: GDB disassembler in Raspberry PI, as we can see the memory is also being deallocated here, there is no difference in the sense that it has to deallocate memory.

4.3 While Loops on Raspberry Pi

In this section we will be testing while loops on the Raspberry Pi.

```
GUTIERREZ_3_WHILE_LOOP.c
1 #include <stdio.h>
2 int main () {
3     int Local_Variable = 0;
4     while( Local_Variable < 10 ) {
5         printf("Local_Variable Value is : %d\n", Local_Variable);
6         Local_Variable++;
7     }
8
9     return 0;
10 }
```

Figure 73: C code for while loop for Raspberry Pi.

```
Reading symbols from test...
(gdb) break main
Breakpoint 1 at 0x1042c: file GUTIERREZ_3_WHILE_LOOP.c, line 3.
(gdb) run
Starting program: /home/pi/Desktop/Take Home Test C/test

Breakpoint 1, main () at GUTIERREZ_3_WHILE_LOOP.c:3
3      int Local_Variable = 0;
(gdb) next 1
4      while( Local_Variable < 10 ) {
```

Figure 74: GDB running a few steps into the program.

```
(gdb) disassemble
Dump of assembler code for function main:
0x00010420 <+0>:    push    {r11, lr}
0x00010424 <+4>:    add     r11, sp, #4
0x00010428 <+8>:    sub     sp, sp, #8
0x0001042c <+12>:   mov     r3, #0
0x00010430 <+16>:   str     r3, [r11, #-8]
=> 0x00010434 <+20>: b       0x10450 <main+48>
0x00010438 <+24>: ldr     r0, [pc, #44] ; 0x1046c <main+76>
0x0001043c <+28>: ldr     r1, [r11, #-8]
0x00010440 <+32>: bl      0x102c8
0x00010444 <+36>: ldr     r3, [r11, #-8]
0x00010448 <+40>: add     r3, r3, #1
0x0001044c <+44>: str     r3, [r11, #-8]
0x00010450 <+48>: ldr     r3, [r11, #-8]
0x00010454 <+52>: cmp     r3, #9
0x00010458 <+56>: ble    0x10438 <main+24>
0x0001045c <+60>: mov     r3, #0
0x00010460 <+64>: mov     r0, r3
0x00010464 <+68>: sub     sp, r11, #4
0x00010468 <+72>: pop    {r11, pc}
0x0001046c <+76>: andeq r0, r1, r4, ror #9
```

Figure 75: GDB disassembly for while loop Raspberry Pi.

In this situation ldr is the jump command for the while loop that will be running the while loop to check the condition. ble is the command for jump if less than which is what we are using in our while loop as we increment our iterator.

```
(gdb) next 1      printf("Local_Variable Value is : %d\n", Local_Variable);
(gdb) next
Local_Variable Value is : 0
6      Local_Variable++;
(gdb) next
4      while( Local_Variable < 10 ) {
(gdb) x &Local_Variable
0x7effef94: 0x00000001
```

Figure 76: GDB output for raspberry pi outputting the value of Local_Variable at the begining of the program.

```
(gdb) x &Local_Variable
0x7effef94: 0x0000000a
```

Figure 77: GDB output for Raspberry pi once the value of Local_Variable is equal to 10.

```
(gdb) disassemble
Dump of assembler code for function main:
0x000010420 <+0>: push    {r11, lr}
0x000010424 <+4>: add     r11, sp, #4
0x000010428 <+8>: sub     sp, sp, #8
0x00001042c <+12>: mov     r3, #0
0x000010430 <+16>: str     r3, [r11, #-8]
0x000010434 <+20>: b       0x10450 <main+48>
0x000010438 <+24>: ldr     r0, [pc, #44] ; 0x1046c <main+76>
0x00001043c <+28>: ldr     r1, [r11, #-8]
0x000010440 <+32>: bl      0x102c8
0x000010444 <+36>: ldr     r3, [r11, #-8]
0x000010448 <+40>: add     r3, r3, #1
0x00001044c <+44>: str     r3, [r11, #-8]
0x000010450 <+48>: ldr     r3, [r11, #-8]
0x000010454 <+52>: cmp     r3, #9
0x000010458 <+56>: ble    0x10438 <main+24>
0x00001045c <+60>: mov     r3, #0
=> 0x000010460 <+64>: mov     r0, r3
0x000010464 <+68>: sub    sp, r11, #4
0x000010468 <+72>: pop    {r11, pc}
0x00001046c <+76>: andeq r0, r1, r4, ror #9
End of assembler dump.
```

Figure 78: GDB disassemle for while loop after termination, after it is finished it is deallocating memory as expected just like it has happened in the previous other compilers and tests as well.

4.4 For Loops on Raspberry Pi

In this section we will be testing for loop on the Raspberry Pi.

```
GUTIERREZ_4_FOR_LOOP.c
1 #include <stdio.h>
2 int main () {
3     int Local_Variable;
4     for( Local_Variable = 10; Local_Variable < 20; Local_Variable = Local_Variable + 1 ){
5         printf("Local_Variable's Value is: %d\n", Local_Variable);
6     }
7     return 0;
8 }
```

Figure 79: C code for for loop in Raspberry pi.

```
(gdb) disassemble
Dump of assembler code for function main:
0x00010420 <+0>:    push    {r11, lr}
0x00010424 <+4>:    add     r11, sp, #4
0x00010428 <+8>:    sub     sp, sp, #8
0x0001042c <+12>:   mov     r3, #10
0x00010430 <+16>:   str     r3, [r11, #-8]
0x00010434 <+20>:   b       0x10450 <main+48>
=> 0x00010438 <+24>: ldr     r0, [pc, #44] ; 0x1046c <main+76>
0x0001043c <+28>:   ldr     r1, [r11, #-8]
0x00010440 <+32>:   bl      0x102c8
0x00010444 <+36>:   ldr     r3, [r11, #-8]
0x00010448 <+40>:   add     r3, r3, #1
0x0001044c <+44>:   str     r3, [r11, #-8]
0x00010450 <+48>:   ldr     r3, [r11, #-8]
0x00010454 <+52>:   cmp     r3, #19
0x00010458 <+56>:   ble    0x10438 <main+24>
0x0001045c <+60>:   mov     r3, #0
0x00010460 <+64>:   mov     r0, r3
0x00010464 <+68>:   sub     sp, r11, #4
0x00010468 <+72>:   pop    {r11, pc}
0x0001046c <+76>:   andeq r0, r1, r4, ror #9
End of assembler dump.
```

Figure 80: GDb Disassembly for Raspberry pi for loop.

```
(gdb) next
Local_Variable's Value is: 15
4         for( Local_Variable = 10; Local_Variable < 20; Local_Variable = Local
_Variable + 1 ){
(gdb) x &Local_Variable
0x7effef94: 0x0000000f
```

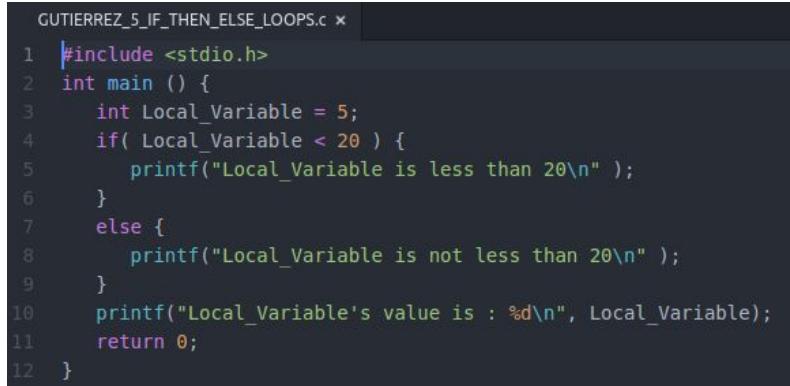
Figure 81: GDB output for the value of the Local_Variable and the address, the address is 0x7effef94 and the value at the address is 15. This is when the program will terminate because now the condition for the for loop to end is met.

```
(gdb) disassemble
Dump of assembler code for function main:
0x00010420 <+0>: push   {r11, lr}
0x00010424 <+4>: add    r11, sp, #4
0x00010428 <+8>: sub    sp, sp, #8
0x0001042c <+12>: mov    r3, #10
0x00010430 <+16>: str    r3, [r11, #-8]
0x00010434 <+20>: b      0x10450 <main+48>
0x00010438 <+24>: ldr    r0, [pc, #44] ; 0x1046c <main+76>
0x0001043c <+28>: ldr    r1, [r11, #-8]
0x00010440 <+32>: bl     0x102c8
0x00010444 <+36>: ldr    r3, [r11, #-8]
0x00010448 <+40>: add    r3, r3, #1
0x0001044c <+44>: str    r3, [r11, #-8]
0x00010450 <+48>: ldr    r3, [r11, #-8]
0x00010454 <+52>: cmp    r3, #19
0x00010458 <+56>: ble   0x10438 <main+24>
0x0001045c <+60>: mov    r3, #0
=> 0x00010460 <+64>: mov    r0, r3
0x00010464 <+68>: sub    sp, r11, #4
0x00010468 <+72>: pop    {r11, pc}
0x0001046c <+76>: andeq r0, r1, r4, ror #9
End of assembler dump.
```

Figure 82: GDB disassembly for Raspberry Pi for lop after the function is finished running. Now the compiler is deallocating memory back to the stack.

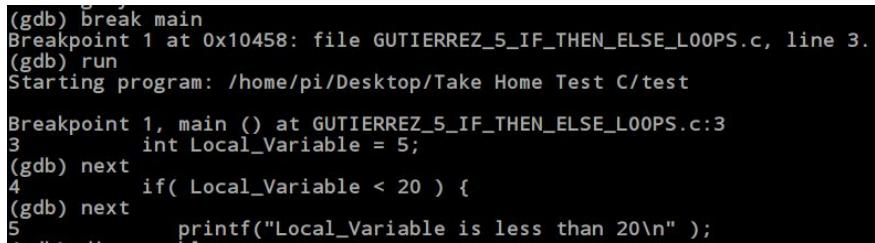
4.5 If-then-else statements on Raspberry Pi

In this section we will be testing if-then-else statements in the Raspberry Pi, we will be checking when a condition is or isn't met and how branching works in the Raspberry Pi.



```
GUTIERREZ_5_IF_THEN_ELSE_LOOPS.c *
1 #include <stdio.h>
2 int main () {
3     int Local_Variable = 5;
4     if( Local_Variable < 20 ) {
5         printf("Local_Variable is less than 20\n" );
6     }
7     else {
8         printf("Local_Variable is not less than 20\n" );
9     }
10    printf("Local_Variable's value is : %d\n", Local_Variable);
11    return 0;
12 }
```

Figure 83: C code for if-then-else case for Raspberry Pi.



```
(gdb) break main
Breakpoint 1 at 0x10458: file GUTIERREZ_5_IF_THEN_ELSE_LOOPS.c, line 3.
(gdb) run
Starting program: /home/pi/Desktop/Take Home Test C/test

Breakpoint 1, main () at GUTIERREZ_5_IF_THEN_ELSE_LOOPS.c:3
3     int Local_Variable = 5;
(gdb) next
4     if( Local_Variable < 20 ) {
(gdb) next
5         printf("Local_Variable is less than 20\n" );
```

Figure 84: GDB output showing the conditions that will be ran through in the if-then-else statement.

```
(gdb) disassemble
Dump of assembler code for function main:
0x0001044c <+0>:    push    {r11, lr}
0x00010450 <+4>:    add     r11, sp, #4
0x00010454 <+8>:    sub     sp, sp, #8
0x00010458 <+12>:   mov     r3, #5
0x0001045c <+16>:   str     r3, [r11, #-8]
0x00010460 <+20>:   ldr     r3, [r11, #-8]
0x00010464 <+24>:   cmp     r3, #19
0x00010468 <+28>:   bgt    0x10478 <main+44>
=> 0x0001046c <+32>: ldr     r0, [pc, #40] ; 0x1049c <main+80>
0x00010470 <+36>:   bl      0x102f4
0x00010474 <+40>:   b       0x10480 <main+52>
0x00010478 <+44>:   ldr     r0, [pc, #32] ; 0x104a0 <main+84>
0x0001047c <+48>:   bl      0x102f4
0x00010480 <+52>:   ldr     r0, [pc, #28] ; 0x104a4 <main+88>
0x00010484 <+56>:   ldr     r1, [r11, #-8]
0x00010488 <+60>:   bl      0x102e8
0x0001048c <+64>:   mov     r3, #0
0x00010490 <+68>:   mov     r0, r3
0x00010494 <+72>:   sub     sp, r11, #4
0x00010498 <+76>:   pop    {r11, pc}
0x0001049c <+80>:   andeq  r0, r1, r12, lsl r5
0x000104a0 <+84>:   andeq  r0, r1, r12, lsr r5
0x000104a4 <+88>:   andeq  r0, r1, r0, ror #10
End of assembler dump.
```

Figure 85: GDB disassembly for if-then-else statements on Raspberry Pi.

```
(gdb) next
Local_Variable is less than 20
10      printf("Local_Variable's value is : %d\n", Local_Variable);
(gdb) next
Local_Variable's value is : 5
11      return 0;
(gdb) next
12 }
```

Figure 86: GDB printing the final values showing that it is less than 20 and that the value is 5.

```
(gdb) disassemble
Dump of assembler code for function main:
0x0001044c <+0>:    push    {r11, lr}
0x00010450 <+4>:    add     r11, sp, #4
0x00010454 <+8>:    sub     sp, sp, #8
0x00010458 <+12>:   mov     r3, #5
0x0001045c <+16>:   str     r3, [r11, #-8]
0x00010460 <+20>:   ldr     r3, [r11, #-8]
0x00010464 <+24>:   cmp     r3, #19
0x00010468 <+28>:   bgt    0x10478 <main+44>
0x0001046c <+32>:   ldr     r0, [pc, #40]    ; 0x1049c <main+80>
0x00010470 <+36>:   bl      0x102f4
0x00010474 <+40>:   b       0x10480 <main+52>
0x00010478 <+44>:   ldr     r0, [pc, #32]    ; 0x104a0 <main+84>
0x0001047c <+48>:   bl      0x102f4
0x00010480 <+52>:   ldr     r0, [pc, #28]    ; 0x104a4 <main+88>
0x00010484 <+56>:   ldr     r1, [r11, #-8]
0x00010488 <+60>:   bl      0x102e8
0x0001048c <+64>:   mov     r3, #0
=> 0x00010490 <+68>:  mov     r0, r3
0x00010494 <+72>:   sub     sp, r11, #4
0x00010498 <+76>:   pop    {r11, pc}
0x0001049c <+80>:   andeq r0, r1, r12, lsl r5
0x000104a0 <+84>:   andeq r0, r1, r12, lsr r5
0x000104a4 <+88>:   andeq r0, r1, r0, ror #10
End of assembler dump.
```

Figure 87: GDB disassembly for if-then else statement again deallocating memory.

4.6 Calling Myadd Function on Raspberry Pi

In this section we will be calling a function called Myadd in the Raspberry Pi.

```
GUTIERREZ_6_CALLING_MY_ADD.c  *
1 int myadd (int& const Word1, int& const Word2) {
2     return Word1 + Word2;
3 }
4 int main() {
5     int Word1 = 0x7fffffff;
6     int Word2 = 0xffffffff;
7     int Word3 = 0;
8     Word3 = myadd(Word1, Word2);
9     return 0;
10 }
```

Figure 88: C code for calling Myadd function on the Raspberry Pi.

```
(gdb) break main
Breakpoint 1 at 0x10424: file GUTIERREZ_6_CALLING_MY_ADD.c, line 5.
(gdb) run
Starting program: /home/pi/Desktop/Take Home Test C/test

Breakpoint 1, main () at GUTIERREZ_6_CALLING_MY_ADD.c:5
5      int Word1 = 0x7fffffff;
(gdb) next
6      int Word2 = 0xffffffff;
(gdb) next
7      int Word3 = 0;
```

Figure 89: GDB output of setting the value of Word1 to the most positive number and Word 2 to -1 and word3 to 0.

```
(gdb) disassemble
Dump of assembler code for function main:
0x00010418 <+0>:    push    {r11, lr}
0x0001041c <+4>:    add     r11, sp, #4
0x00010420 <+8>:    sub     sp, sp, #16
0x00010424 <+12>:   mvn    r3, #-2147483648      ; 0x80000000
0x00010428 <+16>:   str     r3, [r11, #-8]
0x0001042c <+20>:   mvn    r3, #0
0x00010430 <+24>:   str     r3, [r11, #-12]
=> 0x00010434 <+28>:  mov     r3, #0
0x00010438 <+32>:   str     r3, [r11, #-16]
0x0001043c <+36>:   ldr     r0, [r11, #-8]
0x00010440 <+40>:   ldr     r1, [r11, #-12]
0x00010444 <+44>:   bl     0x103e8 <myadd>
0x00010448 <+48>:   str     r0, [r11, #-16]
0x0001044c <+52>:   mov     r3, #0
0x00010450 <+56>:   mov     r0, r3
0x00010454 <+60>:   sub     sp, r11, #4
0x00010458 <+64>:   pop    {r11, pc}
End of assembler dump.
```

Figure 90: GDB disassembly after compiling the code for running myadd function.

```
(gdb) next
8          Word3 = myadd(Word1, Word2);
(gdb) x &Word3
0x7effef8c:    0x00000000
(gdb) next
9          return 0;
(gdb) x &Word3
0x7effef8c:    0x7fffffff
```

Figure 91: GDB output for address and value of Word3.

In this situation we are outputting the value of Word3 before calling the myadd function and after. The address is the same both time, it is 0x7effef8c and the initial value is 0 as we set it to be, after adding the most positive number and negative 1 the value we have is 0x7fffffe which is correct because it is one less than the most positive number.

```
(gdb) disassemble
Dump of assembler code for function main:
0x00010418 <+0>: push   {r11, lr}
0x0001041c <+4>: add    r11, sp, #4
0x00010420 <+8>: sub    sp, sp, #16
0x00010424 <+12>: mvn   r3, #-2147483648      ; 0x80000000
0x00010428 <+16>: str    r3, [r11, #-8]
0x0001042c <+20>: mvn   r3, #0
0x00010430 <+24>: str    r3, [r11, #-12]
0x00010434 <+28>: mov    r3, #0
0x00010438 <+32>: str    r3, [r11, #-16]
0x0001043c <+36>: ldr    r0, [r11, #-8]
0x00010440 <+40>: ldr    r1, [r11, #-12]
0x00010444 <+44>: bl    0x103e8 <myadd>
0x00010448 <+48>: str    r0, [r11, #-16]
=> 0x0001044c <+52>: mov    r3, #0
0x00010450 <+56>: mov    r0, r3
0x00010454 <+60>: sub    sp, r11, #4
0x00010458 <+64>: pop    {r11, pc}
End of assembler dump.
```

Figure 92: GDB Disassembly after terminating the program, as we can see it is also deallocating the stack pointer again in order to release the memory.

In the line with +44 we see the name and address of the function myadd which is how the code was ran instead of using a jump and link or just a jump or jump based on any other kind of condition.

4.7 Logical Operators on Raspberry Pi

In this case we will be running logical operations on Raspberry Pi.

```
GUTIERREZ_7_LOGICAL_OPERATOR.c
1 void main() {
2     static int s0 = 9;
3     static int t1 = 0x3c00;
4     static int t2 = 0xdc0;
5     static int t3 = 0;
6
7     t3 = s0 << 4;
8     static int t0 = 0;
9
10    t0 = t1 & t2;
11
12    t0 = t1 | t2;
13
14    t0 = ~t1;
15 }
```

Figure 93: C code for logical operators that will be run on Raspberry Pi.

```
(gdb) break main
Breakpoint 1 at 0x103f0: file GUTIERREZ_7_LOGICAL_OPERATOR.c, line 7.
(gdb) run
Starting program: /home/pi/Desktop/Take Home Test C/test
nex
Breakpoint 1, main () at GUTIERREZ_7_LOGICAL_OPERATOR.c:7
7         t3 = s0 << 4;
(gdb) next
10        t0 = t1 & t2;
```

Figure 94: GDB output for the first 2 lines and operations.

```
(gdb) x &t3
0x20618 <t3.4134>:      0x00000090
(gdb) x &t0
0x2061c <t0.4135>:      0x00000000
(gdb) next
12        t0 = t1 | t2;
(gdb) x &t0
0x2061c <t0.4135>:      0x0000c00
(gdb) next
14        t0 = ~t1;
(gdb) x &t0
0x2061c <t0.4135>:      0x00003dc0
(gdb) next
15        }
(gdb) x &t0
0x2061c <t0.4135>:      0xfffffc3ff
```

Figure 95: GDB output for running the logical operations on the Raspberry Pi.

The first thing is to make t0 logical or of t1 and t2, then we did logical not on t0 from t1 which is when we get the value 0xfffffcff which is the inverse of 0x00003dc0.

```
(gdb) disassemble
Dump of assembler code for function main:
0x0000103e8 <+0>:    push   {r11}          ; (str r11, [sp, #-4]!)
0x0000103ec <+4>:    add    r11, sp, #0
0x0000103f0 <+8>:    ldr    r3, [pc, #100] ; 0x1045c <main+116>
0x0000103f4 <+12>:   ldr    r3, [r3]
0x0000103f8 <+16>:   lsl    r3, r3, #4
0x0000103fc <+20>:   ldr    r2, [pc, #92] ; 0x10460 <main+120>
0x000010400 <+24>:   str    r3, [r2]
0x000010404 <+28>:   ldr    r3, [pc, #88] ; 0x10464 <main+124>
0x000010408 <+32>:   ldr    r2, [r3]
0x00001040c <+36>:   ldr    r3, [pc, #84] ; 0x10468 <main+128>
0x000010410 <+40>:   ldr    r3, [r3]
0x000010414 <+44>:   and    r3, r3, r2
0x000010418 <+48>:   ldr    r2, [pc, #76] ; 0x1046c <main+132>
0x00001041c <+52>:   str    r3, [r2]
0x000010420 <+56>:   ldr    r3, [pc, #60] ; 0x10464 <main+124>
0x000010424 <+60>:   ldr    r2, [r3]
0x000010428 <+64>:   ldr    r3, [pc, #56] ; 0x10468 <main+128>
0x00001042c <+68>:   ldr    r3, [r3]
0x000010430 <+72>:   orr    r3, r2, r3
0x000010434 <+76>:   ldr    r2, [pc, #48] ; 0x1046c <main+132>
0x000010438 <+80>:   str    r3, [r2]
0x00001043c <+84>:   ldr    r3, [pc, #32] ; 0x10464 <main+124>
0x000010440 <+88>:   ldr    r3, [r3]
0x000010444 <+92>:   mvn    r3, r3
0x000010448 <+96>:   ldr    r2, [pc, #28] ; 0x1046c <main+132>
0x00001044c <+100>:  str    r3, [r2]
=> 0x000010450 <+104>: sub    sp, r11, #0
0x000010454 <+108>: pop    {r11}          ; (ldr r11, [sp], #4)
0x000010458 <+112>: bx    lr
0x00001045c <+116>: andeq r0, r2, r8, lsl #12
0x000010460 <+120>: andeq r0, r2, r8, lsl r6
0x000010464 <+124>: andeq r0, r2, r12, lsl #12
0x000010468 <+128>: andeq r0, r2, r0, lsl r6
0x00001046c <+132>: andeq r0, r2, r12, lsl r6
End of assembler dump.
```

Figure 96: GDB disassembly after running logical operations, as we can see it is deallocating the stack pointer using pop at +108.

4.8 Using Array on Raspberry Pi

In this section we will be using arrays using Raspberry Pi, we will observe what happens in memory and how the program runs properly.

```
GUTIERREZ_8_USING_ARRAY.c  x
1 void main() {
2     static int h = 25;
3     static int A[100];
4     A[8] = 200;
5     A[12] = h + A[8];
6 }
```

Figure 97: C code for using arrays in Raspberry Pi.

```
(gdb) break main
Breakpoint 1 at 0x103f0: file GUTIERREZ_8_USING_ARRAY.c, line 4.
(gdb) run
Starting program: /home/pi/Desktop/Take Home Test C/test

Breakpoint 1, main () at GUTIERREZ_8_USING_ARRAY.c:4
4      A[8] = 200;
(gdb) next
5      A[12] = h + A[8];
```

Figure 98: GDB output showing a few lines into the code after using the array in Raspberry Pi.

```
(gdb) disassemble
Dump of assembler code for function main:
0x0000103e8 <+0>: push   {r11}          ; (str r11, [sp, #-4]!)
0x0000103ec <+4>: add    r11, sp, #0
0x0000103f0 <+8>: ldr    r3, [pc, #44]   ; 0x10424 <main+60>
0x0000103f4 <+12>: mov    r2, #200       ; 0xc8
0x0000103f8 <+16>: str    r2, [r3, #32]
=> 0x0000103fc <+20>: ldr    r3, [pc, #32]   ; 0x10424 <main+60>
0x000010400 <+24>: ldr    r2, [r3, #32]
0x000010404 <+28>: ldr    r3, [pc, #28]   ; 0x10428 <main+64>
0x000010408 <+32>: ldr    r3, [r3]
0x00001040c <+36>: add    r3, r2, r3
0x000010410 <+40>: ldr    r2, [pc, #12]   ; 0x10424 <main+60>
0x000010414 <+44>: str    r3, [r2, #48]   ; 0x30
0x000010418 <+48>: sub    sp, r11, #0
0x00001041c <+52>: pop    {r11}          ; (ldr r11, [sp], #4)
0x000010420 <+56>: bx    lr
0x000010424 <+60>: andeq r0, r2, r12, asr #11
0x000010428 <+64>: andeq r0, r2, r4, asr #11
End of assembler dump.
```

Figure 99: GDB disassembly as we can see the value of the array is already stored in memory

because it is static.

The offset from the base pointer is where the value of h will be added in the array of 100 elements and it will be accessed even though other parts of the array are not even going to be initialized by default. That does not cause a problem because it is allowed, it is allowed because it is an accessible address. And that is why the offsets are very large and printed on the side in

order to help the user understand what is going on with the program when writing to and accessing the elements in an array.

```
(gdb) next
6
(gdb) disassemble
Dump of assembler code for function main:
0x0000103e8 <+0>: push   {r11}          ; (str r11, [sp, #-4]!)
0x0000103ec <+4>: add    r11, sp, #0
0x0000103f0 <+8>: ldr    r3, [pc, #44]   ; 0x10424 <main+60>
0x0000103f4 <+12>: mov    r2, #200        ; 0xc8
0x0000103f8 <+16>: str    r2, [r3, #32]
0x0000103fc <+20>: ldr    r3, [pc, #32]   ; 0x10424 <main+60>
0x000010400 <+24>: ldr    r2, [r3, #32]
0x000010404 <+28>: ldr    r3, [pc, #28]   ; 0x10428 <main+64>
0x000010408 <+32>: ldr    r3, [r3]
0x00001040c <+36>: add    r3, r2, r3
0x000010410 <+40>: ldr    r2, [pc, #12]   ; 0x10424 <main+60>
0x000010414 <+44>: str    r3, [r2, #48]   ; 0x30
=> 0x000010418 <+48>: sub    sp, r11, #0
0x00001041c <+52>: pop    {r11}          ; (ldr r11, [sp], #4)
0x000010420 <+56>: bx    lr
0x000010424 <+60>: andeq r0, r2, r12, asr #11
0x000010428 <+64>: andeq r0, r2, r4, asr #11
End of assembler dump.
```

Figure 100: GDB exiting after using the array again deallocating memory on Raspberry Pi.

PART 4 WINDOWS 32 BIT COMPILER

The purpose of this section is to test the way that local variables, static variables, while loops, for loops, if-then-else statements, calling a function, logical operators, and using arrays is handled on Windows using a 32 bit compiler and debugger on Visual Studio. We will be testing simple code for each of the 8 cases and we will be observing how the variables are handled as well as how data is stored and accessed. We will also be noting what is similar and different when compared to the other 3 platforms that will be observed during this exam. We will also note that Windows 32 bit compiler uses Big Endian, that means it stores the most significant byte of information in memory into the lowest address.

5.1 Local Variables on Windows 32 Bit Compiler

In this section we will be observing how the windows 32 bit compiler interacts stores and uses local variables in memory. We will be using c++ code and visual studio in order to test, we will also be using the debugger to make our observations.

```

1 // // ConsoleApplication1.cpp : Defines the entry point for the console application.
2 [
3
4     #include "stdafx.h"
5     //Using 5 Local Variables to test Local Variables in C
6     int main() {
7         int a = 1;
8         int b = 2; //ms elapsed
9         int c = 3;
10        int d = 4;
11        int e = 5;
12        a = b + c;
13        d = a - e;
14        return 0;
15    }
16 ]

```

Figure 101: C++ code for using local variables that will be tested on the windows 32 bit compiler.

```

#include "stdafx.h"
//Using 5 Local Variables to test Local Variables in C
int main() {
012B16A0 push    ebp
012B16A1 mov     ebp,esp
012B16A3 sub    esp,0FCh
012B16A9 push    ebx
012B16AA push    esi
012B16AB push    edi
012B16AC lea     edi,[ebp-0FCh]
012B16B2 mov     ecx,3Fh
012B16B7 mov     eax,0CCCCCCCCh
012B16BC rep stos dword ptr es:[edi]
    int a = 1;
012B16BE mov     dword ptr [a],1
    int b = 2;
012B16C5 mov     dword ptr [b],2
    int c = 3;
012B16CC mov     dword ptr [c],3
    int d = 4;
012B16D3 mov     dword ptr [d],4
    int e = 5;
012B16DA mov     dword ptr [e],5

```

Figure 102: Disassembly window for using Local variables in Windows 32 bit compiler.

As we can notice the instructions on here are different than the ones on linux or MIPS. Different here we can see that the first instruction at 0x012b16a0 is to push ebp to the stack in this case

that is the base pointer, and the next instruction is to set the stack pointer equal to the base pointer before that happens none of the two pointers have a usable address until they are initialized. The command that says mov dword ptr [a],1 means that I am storing a 32 bit word labeled a and its value is 1, the same respectively continues for the following lines as it stores the additional variables and their respective values.

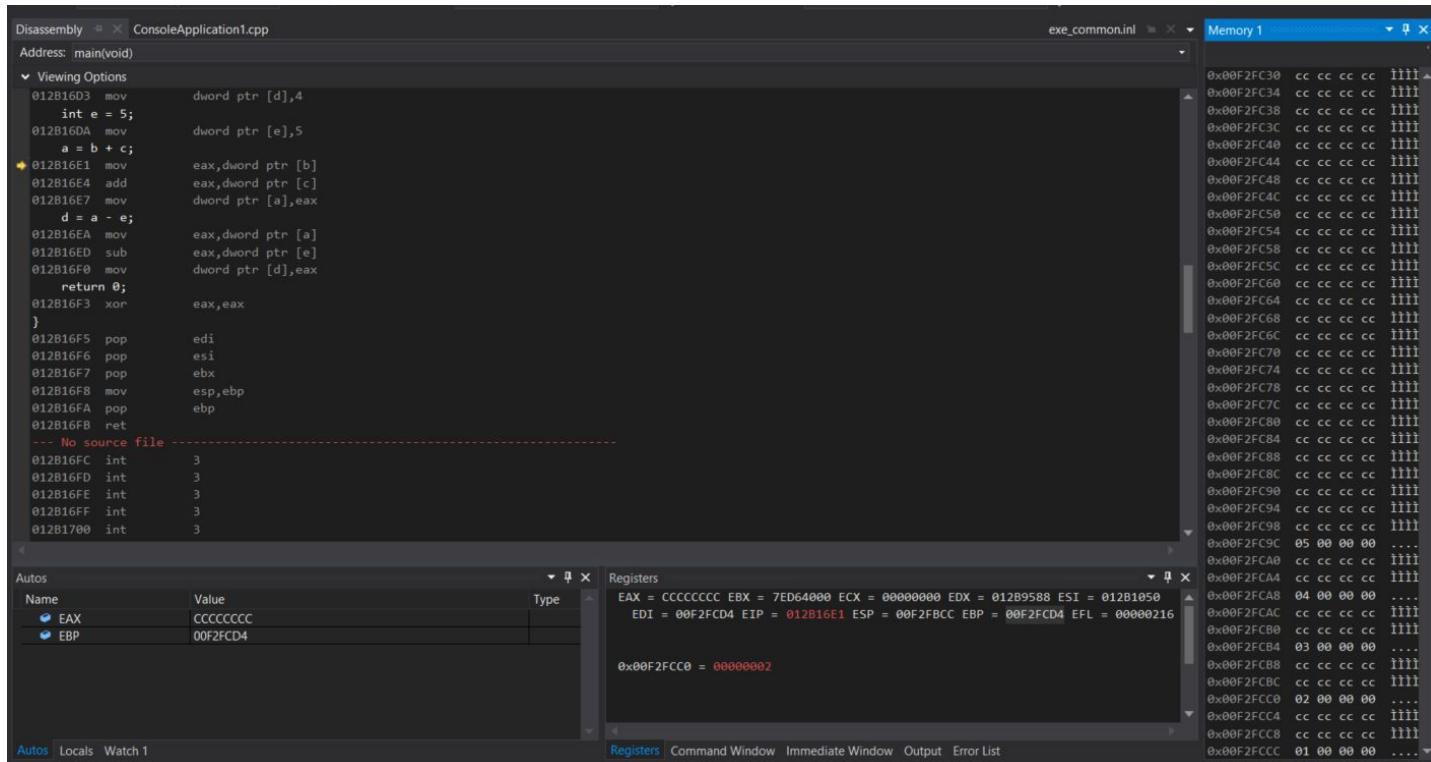


Figure 103: Visual studio windows after initialize some of the local variables into memory.

As we can see in the yellow arrow the current information is being pushed to the EAX in order for it to be used. As we see it the right in the memory window the address 0x00f2fc9c is storing the value of 5 which is our variable E and it is stored in Big endian. Intel in this case is using big endian as we have mentioned before to store data, that means that the most significant byte of information is being stored in memory into the smallest address first. The current address of the base pointer is 0x00f2fcd4 and the address of the current address is 0x012016e1. We can see in

the memory window to the right that the values of our variables are stored there with an offset to the stack pointer and they are then stored using big endian.

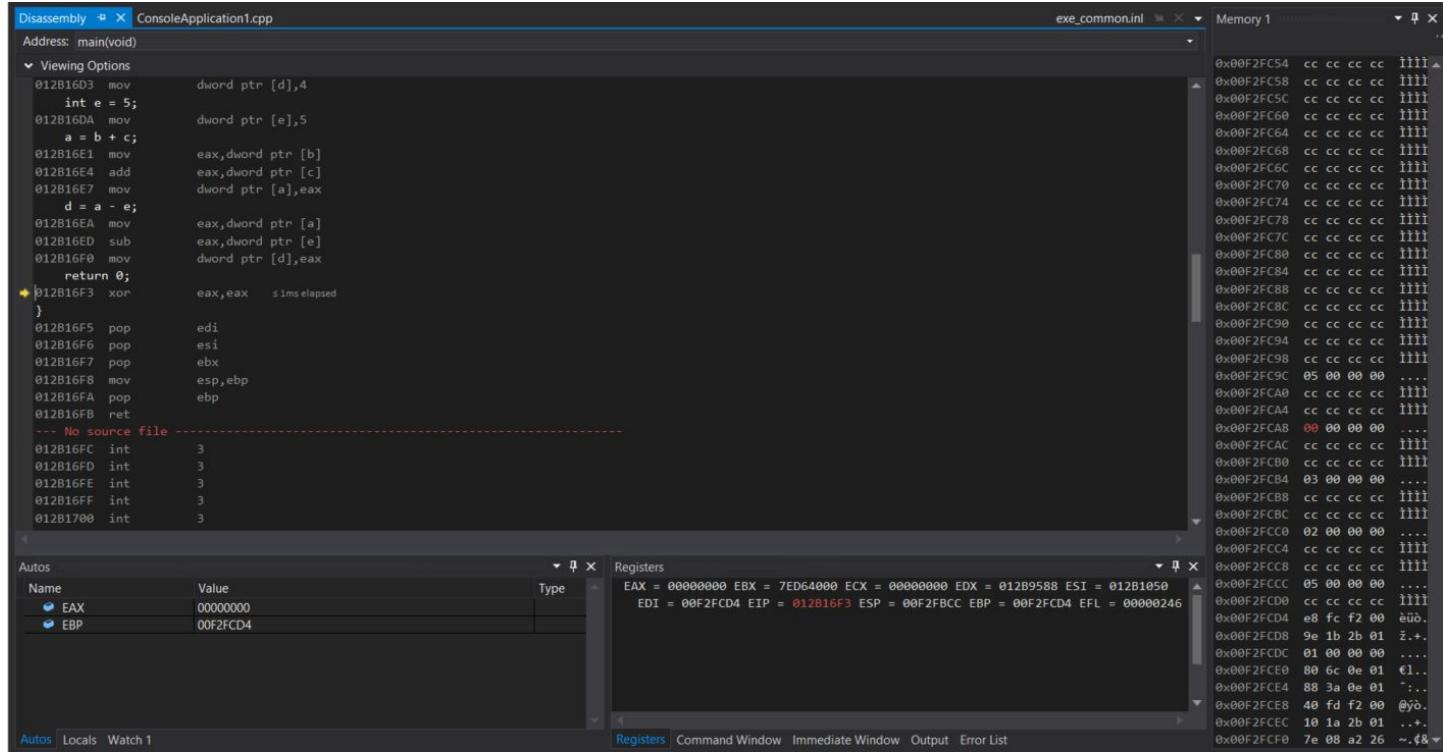
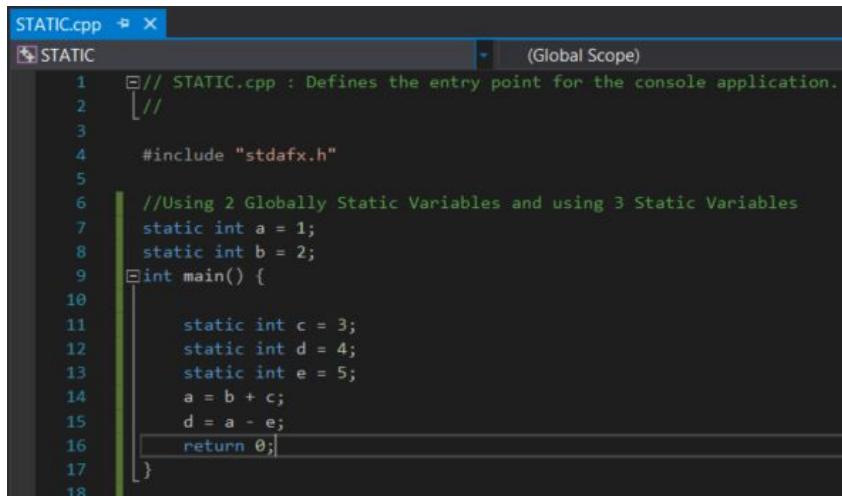


Figure 104: Visual Studio after terminating the using local variables program on windows 32 bit compiler.

As we can see the address where the final operation was performed is 0x00f2fca8 and as it is highlighted in red the value after the operation is 00 and it is stored in reverse order because it is using big endian to store information. The address 0x00f2fca8 is also the address of the variable d. In the final lines of the disassembly code we can see that the program is performing pop on edi, esi, ebx, esp and ebp in that order, that is done to deallocate memory which functions in a similar way that the previous 3 processors do to free memory after a program has been terminated.

5.2 Static Variables on Windows 32 Bit Compiler

In this section we will be observing and testing how windows 32 bit compiler stores and uses static variables in memory.



The screenshot shows the Visual Studio code editor with a file named "STATIC.cpp". The code defines a global scope for five static integers: a, b, c, d, and e. It initializes them with values 1, 2, 3, 4, and 5 respectively. Then it performs simple arithmetic operations: a = b + c, d = a - e, and returns 0. The code is annotated with comments explaining its purpose.

```
// STATIC.cpp : Defines the entry point for the console application.
// Using 2 Globally Static Variables and using 3 Static Variables
static int a = 1;
static int b = 2;
int main() {
    static int c = 3;
    static int d = 4;
    static int e = 5;
    a = b + c;
    d = a - e;
    return 0;
}
```

Figure 105: C++ code for windows 32 bit compiler on visual studio that will be storing 5 static variables into memory and then be using them to perform simple additions and subtraction operations.

```

Disassembly  X STATIC.cpp
Address: main(void)
Viewing Options
//Using 2 Globally Static Variables and using 3 Static Variables
static int a = 1;
static int b = 2;
int main() {
    push    ebp
    mov     ebp,esp
    sub    esp,0C0h
    push    ebx
    push    esi
    push    edi
    lea     edi,[ebp-0C0h]
    mov     ecx,30h
    mov     eax,0CCCCCCCCh
    rep stos dword ptr es:[edi]

    static int c = 3;
    static int d = 4;
    static int e = 5;
    a = b + c;
    mov     eax,dword ptr [b (0C29004h)]
    add     eax,dword ptr ds:[0C29008h]
    mov     dword ptr [a (0C29000h)],eax
    d = a - e;
    mov     eax,dword ptr [a (0C29000h)]
    sub     eax,dword ptr ds:[0C29010h]
    mov     dword ptr ds:[00C2900Ch],eax
    return 0;
    xor    eax,eax
}
    pop    edi
    pop    esi
    pop    ebx
    mov    esp,ebp
    pop    ebp
    ret

```

Figure 106: Disassembly windows from visual studio using static variables on windows 32 bit

compiler.

As we can see just like when we have used static variables in Linux 64 bit compiler and Raspberry Pi the offset from the base pointer is a very large value in order to use static variables. In this case however the value we see in for example eax, dword ptr[a (0x0c29000)],eax is not an offset it is an actual address. The reason we are seeing a complete address in this line of disassembly and for all the other instances of storing static variables is because windows 32 bit compiler is very efficient in storing static variables, instead of taking an offset to instruction

pointer or to the base pointer it uses the literal address of any given variable to access it because it does not have to perform any operations with subtraction with an offset to get to the variable. It has direct access to the variable more efficiently without having to perform any operation, it immediately looks to the address of the variable whenever it is being used after it has already been stored and initialized.

The screenshot shows the Visual Studio IDE interface with the following windows:

- Disassembly**: Shows assembly code for the `main` function. The code initializes static variables `c`, `d`, and `e` to 3, 4, and 5 respectively. It then calculates `d = a - e` and returns the value of `a`.
- Registers**: Displays register values: EAX = 00000005, EBX = 7F5A3000, ECX = 00000000, EDX = 00C295A0, ESI = 00C21050, EDI = 00D2F838, EIP = 00C216C9, ESP = 00D2F76C, EBP = 00D2F838, and EFL = 00000206.
- Memory**: A large window showing memory starting at address 0x00C28F78. Addresses from 0x00C29000 to 0x00C29014 are circled in blue, indicating the addresses of static variables `a`, `b`, `c`, `d`, and `e`.
- Autos**, **Locals**, and **Watch 1**: These windows are visible at the bottom left but are currently empty.

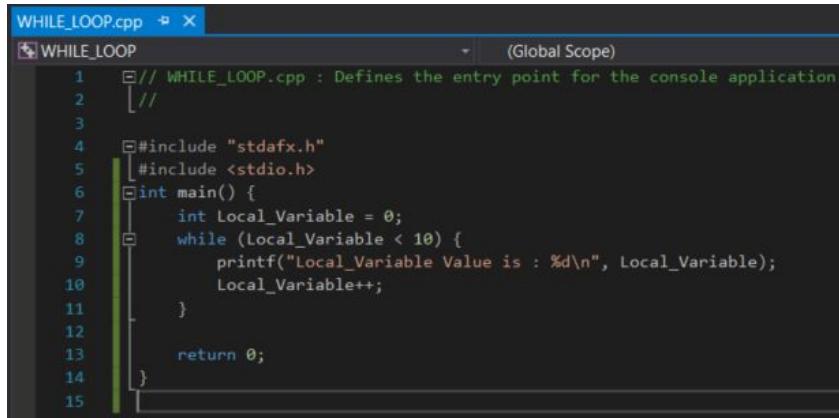
Figure 107: Visual studio windows after running a few steps into a program that is using static variables in memory.

As we can see based on the address circled in the bottom right corner those are the addresses of the variables that are static in memory; a,b,c,d,e. The important thing to note here is that none of those 5 addresses are based on an offset to the base pointer or the instruction pointer that is because they are given a unique address at an offset from either the instruction pointer or the base pointer and that value is large in either case. One way to think about it is that there is some

sort of global pointer that is halfway between the instruction pointer and the base pointer like in MIPS that is closer to the address of static variables in any windows 32 bit compiler like in this case using visual studio.

5.3 While Loop on Windows 32 Bit Compiler

In this section we will be testing and observing how while loops are working in windows 32 bit compiler and compare the differences that it has from linux 64 bit compiler and the raspberry pi and mips.



A screenshot of the Visual Studio code editor showing a file named 'WHILE_LOOP.cpp'. The code is a simple C++ program that defines the entry point for a console application. It includes stdafx.h and stdio.h, and contains a main function that prints the value of a Local_Variable variable until it reaches 10. The code is numbered from 1 to 15.

```
WHILE_LOOP.cpp # X
WHILE_LOOP (Global Scope)
1 // WHILE_LOOP.cpp : Defines the entry point for the console application.
2 [
3
4 #include "stdafx.h"
5 #include <stdio.h>
6 int main() {
7     int Local_Variable = 0;
8     while (Local_Variable < 10) {
9         printf("Local_Variable Value is : %d\n", Local_Variable);
10        Local_Variable++;
11    }
12
13    return 0;
14 }
15 ]
```

Figure 108: C++ code for a while loop that will be tested on visual studio in a 32 bit windows compiler, the condition in this loop is to set a Local_Variable to 0 and then to continue to increment it until we reach the value where the condition that the variable is less than 10 is no longer true.

```

int main() {
011D17E0 push    ebp
011D17E1 mov     ebp,esp
011D17E3 sub    esp,0CCh
011D17E9 push    ebx
011D17EA push    esi
011D17EB push    edi
011D17EC lea     edi,[ebp-0CCh]
011D17F2 mov     ecx,33h
011D17F7 mov     eax,0CCCCCCCCh
011D17FC rep stos  dword ptr es:[edi]
    int Local_Variable = 0;
011D17FE mov     dword ptr [Local_Variable],0
    while (Local_Variable < 10) {
011D1805 cmp     dword ptr [Local_Variable],0Ah
011D1809 jge     main+47h (011D1827h)
        printf("Local_Variable Value is : %d\n", Local_Variable);
011D180B mov     eax,dword ptr [Local_Variable]
011D180E push    eax
011D180F push    offset string "Local_Variable Value is : %d\n" (011D7B30h)
011D1814 call    _printf (011D1334h)
011D1819 add    esp,8
        Local_Variable++;
011D181C mov     eax,dword ptr [Local_Variable]
011D181F add    eax,1
011D1822 mov     dword ptr [Local_Variable],eax
    }
|011D1825 jmp     main+25h (011D1805h)

    return 0;
}

```

Figure 109: Disassembly code for while loop on windows 32 bit compiler.

Instead of using jal, or jmp, or ja, or j, or bne and beq like we have seen previously in linux 64 bit compiler and raspberry pi and mips the command being used here are jge and jmp. Jge will jump function like jump does based on a condition. It basically means jump if a condition is met and that is what we expect to happen. Jmp stands for jump which basically means to jump back to a given address in this case the address of the while loop if it isn't already exited.

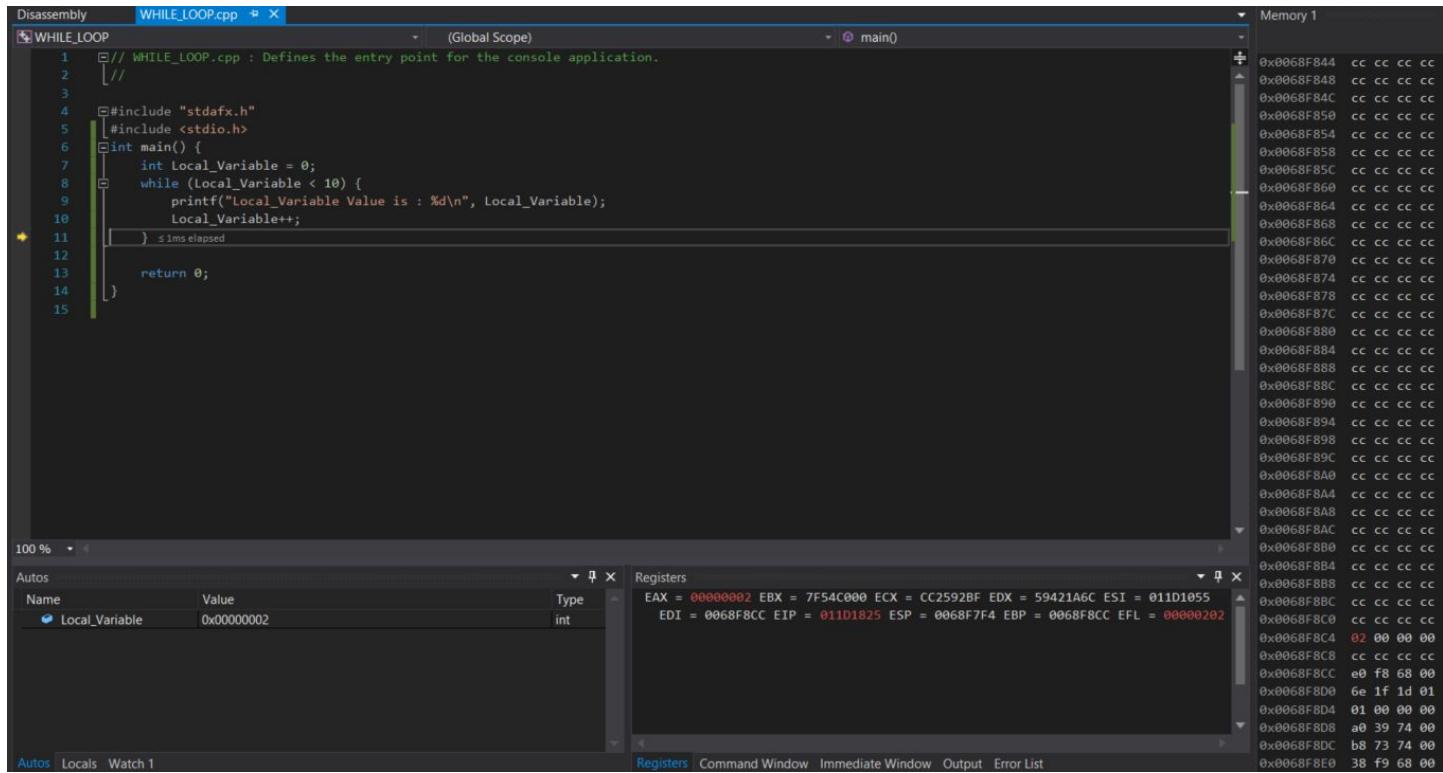


Figure 110: Visual Studio after iterating through the while loop twice, the value at the address 0x0068F8C4 which is currently and it is our iterator value.

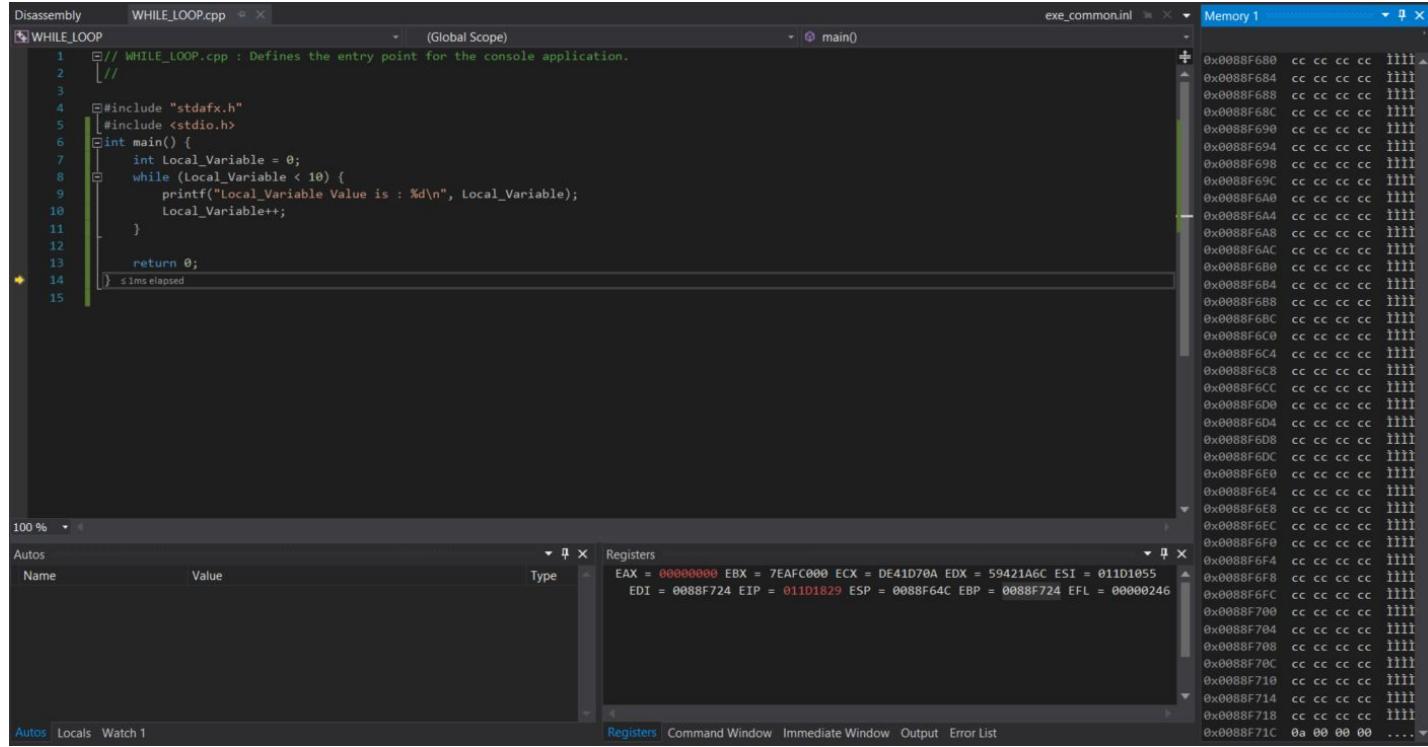


Figure 111: Visual studio, the value for our iterator variable which is Local_variable is now 10

and that is when the while loop will exit now and be finished.

```

Local_Variable Value is : 0
Local_Variable Value is : 1
Local_Variable Value is : 2
Local_Variable Value is : 3
Local_Variable Value is : 4
Local_Variable Value is : 5
Local_Variable Value is : 6
Local_Variable Value is : 7
Local_Variable Value is : 8
Local_Variable Value is : 9

```

Figure 112: Terminal output of the value of the Local_variable as we go through the while loop to show that the value is changing correctly and that the while loop is only exited based upon the proper condition, when the condition for the while loop is no longer met, the program will exit the while loop and it will continue with whatever is next in the code, in this case it will terminate the entire program.

5.4 For Loop on Windows 32 Bit Compiler

In this section we will be looking at for loops in the windows 32 bit compiler.

```

FOR.cpp * X
FOR
  // FOR.cpp : Defines the entry point for the console application.

  [//]

  #include "stdafx.h"
  #include <stdio.h>
  int main() {
    int Local_Variable;
    for (Local_Variable = 10; Local_Variable < 20; Local_Variable = Local_Variable + 1) {
      printf("Local_Variable's Value is: %d\n", Local_Variable);
    }
    return 0;
}

```

Figure 113: C++ code for for loop that will be run in visual studio in the windows 32 bit compiler.

```

#include "stdafx.h"
#include <stdio.h>
int main() {
00ED17E0 push      ebp
00ED17E1 mov       ebp,esp
00ED17E3 sub       esp,0CCh
00ED17E9 push      ebx
00ED17EA push      esi
00ED17EB push      edi
00ED17EC lea       edi,[ebp-0CCh]
00ED17F2 mov       ecx,33h
00ED17F7 mov       eax,0CCCCCCCCh
00ED17FC rep stos   dword ptr es:[edi]
    int Local_Variable;
    for (Local_Variable = 10; Local_Variable < 20; Local_Variable = Local_Variable + 1) {
00ED17FE mov       dword ptr [Local_Variable],0Ah
    int Local_Variable;
    for (Local_Variable = 10; Local_Variable < 20; Local_Variable = Local_Variable + 1) {
00ED1805 jmp      main+30h (0ED1810h)
00ED1807 mov       eax,dword ptr [Local_Variable]
00ED180A add       eax,1
00ED180D mov       dword ptr [Local_Variable],eax
00ED1810 cmp       dword ptr [Local_Variable],14h
00ED1814 jge      main+49h (0ED1829h)
    printf("Local_Variable's Value is: %d\n", Local_Variable);
00ED1816 mov       eax,dword ptr [Local_Variable]
00ED1819 push     eax
00ED181A push     offset string "Local_Variable's Value is: %d\n" (0ED7B30h)
00ED181F call     _printf (0ED1334h)
00ED1824 add       esp,8
    }
00ED1827 jmp      main+27h (0ED1807h)
    return 0;
00ED1829 xor       eax,eax
}

```

Figure 114: Assembly code for windows 32 bit compiler for running a for loop. As we can see again we are using the commands jge and jmp which is going to be used in this case because we have an iterator in our for loop.

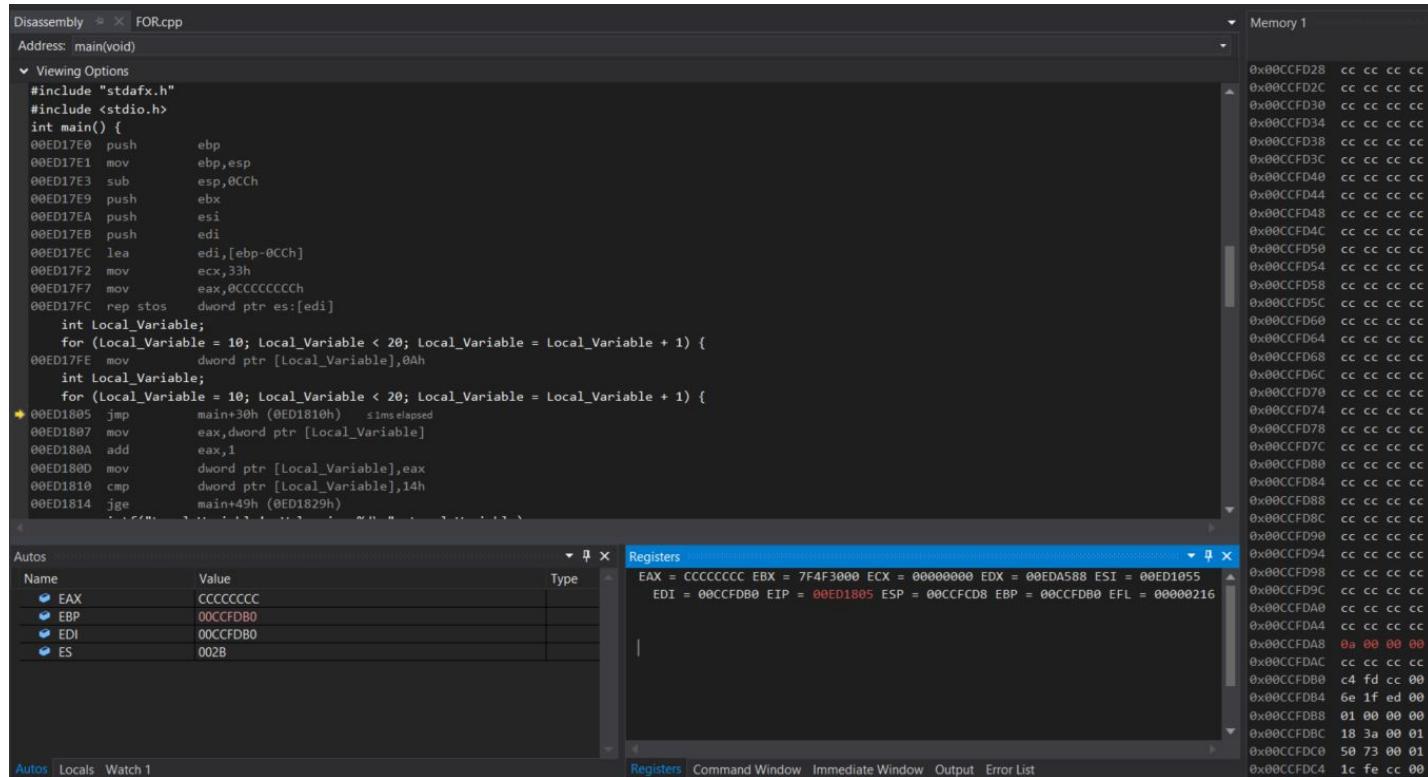


Figure 115: visual studio after running through the program and reaching the value of the for loop in the address 0x0ccfd80 which means that the program was able to successfully jump through the for loop back into main using jmp and jge in order to continue to run the for loop until the condition is met, it increase the value by 1 until reaching a value less than 20 and after that it exists because it is done with the program and no longer needs to jump back into the for loop.

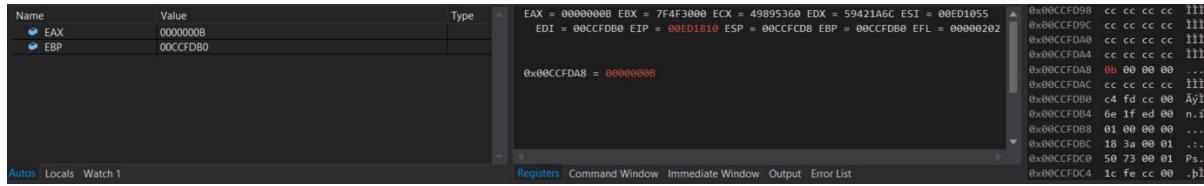


Figure 116: The final image of Visual Studio for when the for loop is finished running and the address of the iterator value at the address 0x00ccfd8 which is the same address of our Local_Variable from the previously mentioned image.

5.5 If-then-else statements on Windows 32 Bit Compiler

In this section we will be running if-then-else statements on the windows 32 bit compiler and observe how it functions properly and what are the commands that it is using to check conditions and to branch into cases.

```

IF_THEN_ELSE.cpp  ✘ X
IF_THEN_ELSE
(Global Scope)

1 // IF_THEN_ELSE.cpp : Defines the entry point for the console application.
2 [
3
4 #include "stdafx.h"
5
6 #include <stdio.h>
7 int main()
8 {
9     int Local_Variable = 5;
10    if (Local_Variable < 20) {
11        printf("Local_Variable is less than 20\n");
12    }
13    else {
14        printf("Local_Variable is not less than 20\n");
15    }
16    printf("Local_Variable's value is : %d\n", Local_Variable);
17 }
18

```

Figure 117: C++ code for if else statement that checks if a variable meets a given condition and then if it does it runs through the if loop if it doesn't then it runs into the else loop.

```

int main() {
013617E0 push    ebp
013617E1 mov     ebp,esp
013617E3 sub    esp,0CCh
013617E9 push    ebx
013617EA push    esi
013617EB push    edi
013617EC lea     edi,[ebp-0CCh]
013617F2 mov     ecx,33h
013617F7 mov     eax,0CCCCCCCCh
013617FC rep stos  dword ptr es:[edi]
    int Local_Variable = 5;
013617FE mov     dword ptr [Local_Variable],5
    if (Local_Variable < 20) {
01361805 cmp     dword ptr [Local_Variable],14h
01361809 jge     main+3Ah (0136181Ah)
        printf("Local_Variable is less than 20\n");
0136180B push    offset string "Local_Variable is less than 20\n" (01367B30h)
01361810 call    _printf (01361334h)
01361815 add    esp,4
    }
    else {
01361818 jmp     main+47h (01361827h)
        printf("Local_Variable is not less than 20\n");
0136181A push    offset string "Local_Variable is not less than ..." (01367B58h)
0136181F call    _printf (01361334h)
01361824 add    esp,4
    }
    printf("Local_Variable's value is : %d\n", Local_Variable);
01361827 mov     eax,dword ptr [Local_Variable]
0136182A push    eax
0136182B push    offset string "Local_Variable's value is : %d\n" (01367B884h)
01361830 call    _printf (01361334h)
}

```

Figure 118: Disassembly code for if else statement in c++ for windows 32 bit compiler.

As we can see again the conditions are checked and met using jge and jmp like in the previous cases that we were checking like the for loop and the while loop and this is still different from the other 3 compilers, linux 64 bit, raspberry pi and MIPS. Now we know that the windows 32 bit compiler uses a library to check conditions that are useful and effective and efficient for the processor that make it simpler for the compiler to know how to check a while loop, for loop or if else condition.

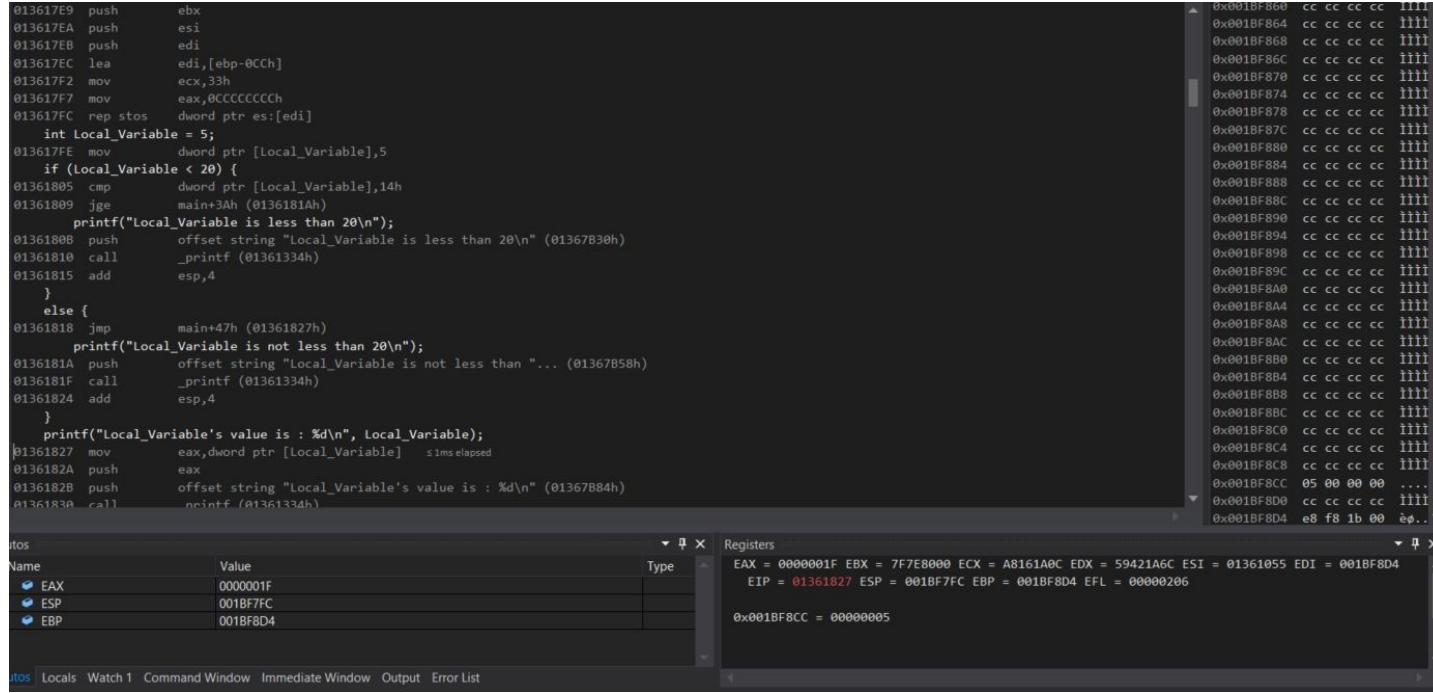


Figure 119: Visual studio after running setting Local_Variable equal to 5 and then noting that the value is less than 20, which means that it will go to the else condition, it will jump to that address in memory in order to run the next operation.

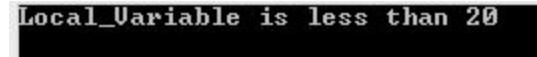


Figure 120: Output showing that the condition is met correctly and that even in a windows 32 bit compiler the program knows how check conditions using jump methods like the other 3 compilers can do.

5.6 Calling myadd function on Windows 32 Bit Compiler

In this section we will be calling a function that adds two values, we can do this without a function but we want to observe what sort of commands or how function calls are made in a windows 32 bit compiler.

```

CALLING_MY_ADD.cpp  X
CALLING_MY_ADD      (Global Scope)

1 //// CALLING_MY_ADD.cpp : Defines the entry point for the console application.
2 [
3
4 #include "stdafx.h"
5
6 int myadd(int& const Word1, int& const Word2) {
7     return Word1 + Word2;
8 }
9 int main() {
10    int Word1 = 0x7fffffff;
11    int Word2 = 0xffffffff;
12    int Word3 = 0;
13    Word3 = myadd(Word1, Word2);
14    return 0;
15 }

```

Figure 121: C++ code for calling my add function that will be run and observed in a windows 32 bit compiler.

```

int main() {
00D41710 push    ebp
00D41711 mov     ebp,esp
00D41713 sub    esp,0E8h
00D41719 push    ebx
00D4171A push    esi
00D4171B push    edi
00D4171C lea     edi,[ebp-0E8h]
00D41722 mov     ecx,3Ah
00D41727 mov     eax,0CCCCCCCCh
00D4172C rep stos dword ptr es:[edi]
00D4172E mov     eax,dword ptr [_security_cookie (0D49000h)]
00D41733 xor     eax,ebp
00D41735 mov     dword ptr [ebp-4],eax
    int Word1 = 0x7fffffff;
00D41738 mov     dword ptr [Word1],7FFFFFFFh
    int Word2 = 0xffffffff;
00D4173F mov     dword ptr [Word2],0FFFFFFFh
    int Word3 = 0;
00D41746 mov     dword ptr [Word3],0
    Word3 = myadd(Word1, Word2);
00D4174D lea     eax,[Word2]
00D41750 push    eax
00D41751 lea     ecx,[Word1]
00D41754 push    ecx
00D41755 call    myadd (0D412A8h)
00D4175A add    esp,8
00D4175D mov     dword ptr [Word3],eax
    return 0;
00D41760 xor     eax,eax
}

```

Figure 122: Disassembly code for calling myadd function using windows 32 bit compiler.

As we can see the disassembly code shows that in windows 32 bit compiler the code that is used is called “call” it calls the function and takes as an operand the address of the function that will be called. The address in this case is 0x0d412a8h which is why the line shows that it is calling the function using an address. Even before that we can see that it is pushing the parameters into the stack from right to left before calling the function in order to dereference after running the program and return to the main function and continue what is in it which in this case is just exiting the function.

The screenshot shows the Visual Studio debugger interface with three main windows:

- Disassembly:** Shows the assembly code for the `main()` function. The assembly instructions are color-coded by category. Key instructions include:
 - Pushes `ebp` onto the stack.
 - Loads `ebp` into `esp`.
 - Subtracts `0E8h` from `esp`.
 - Pushes `ebx` onto the stack.
 - Pushes `esi` onto the stack.
 - Pushes `edi` onto the stack.
 - Loads `edi` into `[ebp-0E8h]`.
 - Loads `ecx` into `3Ah`.
 - Loads `eax` into `0CCCCCCCCh`.
 - Performs a `rep stos` operation.
 - Loads `eax` into `dword ptr [__security_cookie (0D49000h)]`.
 - Performs a `xor eax, eax` operation.
 - Performs a `mov eax, [ebp]` operation.
 - Loads `eax` into `dword ptr [ebp-4]`.
 - Initializes `Word1` to `0xffffffff`.
 - Initializes `Word2` to `0xffffffff`.
 - Initializes `Word3` to `0`.
 - Calls `myadd(Word1, Word2)`.
 - Loads `eax` into `[Word2]`.
 - Pushes `eax` onto the stack.
 - Loads `ecx` into `[Word1]`.
 - Pushes `ecx` onto the stack.
 - Calls `myadd (0D412A8h)`.
 - Adds `esp` to `8`.
- Registers:** Shows the current register values:
 - `EAX = F47B0799`
 - `EBX = 7EDAE000`
 - `ECX = 00000000`
 - `EDX = 00D49588`
 - `ESI = 00D41055`
 - `EDI = 0091F9B4`
 - `EIP = 00D41746`
 - `ESP = 0091F8C0`
 - `EBP = 0091F9B4`
 - `EFL = 00000286`
- Memory:** Shows the memory dump starting at `0x0091F940`, where `eax` is set to `CCCCCCCC`.

Figure 123: Visual studio registers, memory window and disassembly code after launching the 32 bit windows debugger.

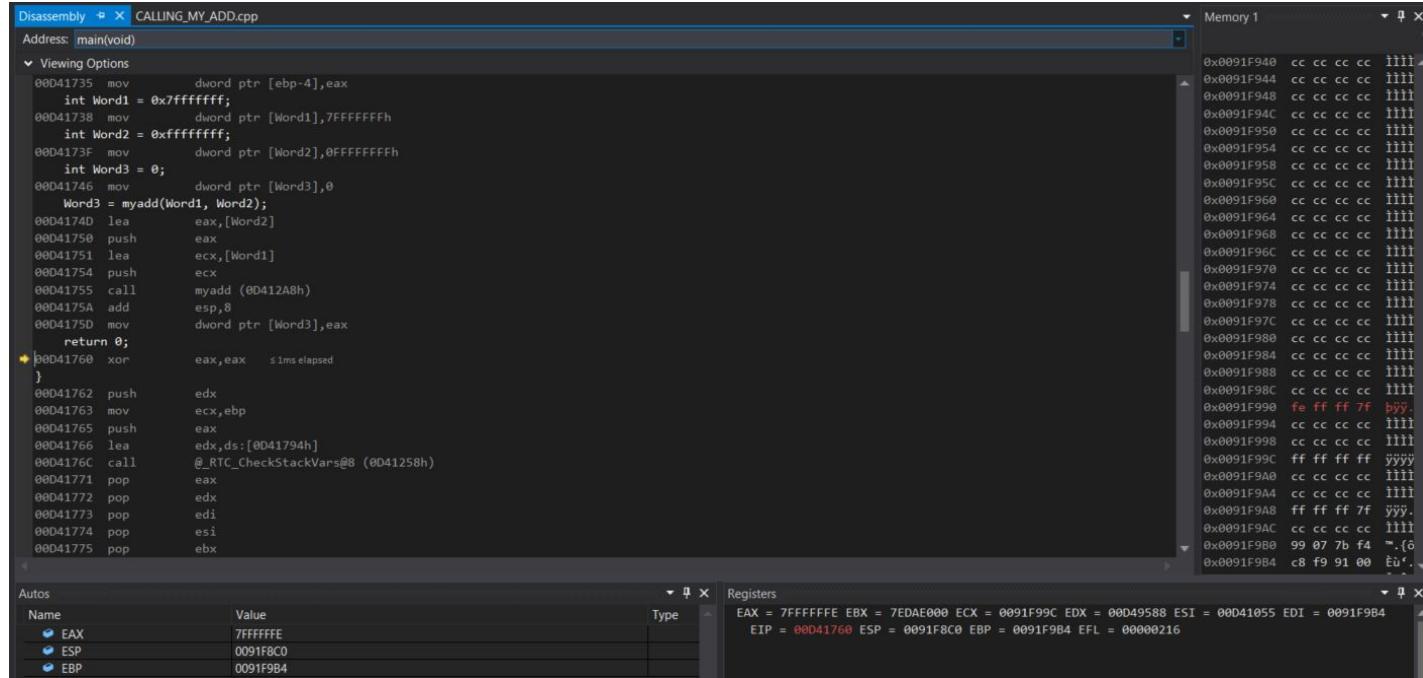


Figure 124: Visual studio after completing the call of the myadd function. We can notice that at the address 0x0091f990 the value stored is feffff7f which is a value stored in big endian notation which means the literal value is 7ffffffe which is one less than the most positive number that can be stored in 32 bits. Which means our program ran and terminated successfully because we were subtracting 1 from the largest positive number that could be stored in 32 bits.

5.7 Logical Operators on Windows 32 Bit Compiler

In this section we will be running logical operations on bits of information on a windows 32 bit compiler in order to compare how they work different from MIPS Linux 64 bit and the raspberry pi.

```

LOGICAL_OPERATOR.cpp  ✘ X
LOGICAL_OPERATOR
1 // LOGICAL_OPERATOR.cpp : Defines the entry point for the console application.
2 [
3
4 #include "stdafx.h"
5 int main() {
6     int s0 = 9;
7     int t1 = 0x3c00;
8     int t2 = 0xdc0;
9     int t3 = 0;
10
11     t3 = s0 << 4;
12     int t0 = 0;
13
14     t0 = t1 & t2;
15
16     t0 = t1 | t2;
17
18     t0 = ~t1;
19     return 0;
20 }

```

Figure 125: C++ code for logical operations that will be run on windows 32 bit compiler.

```

int s0 = 9;
010516BE mov        dword ptr [s0],9
int t1 = 0x3c00;
010516C5 mov        dword ptr [t1],3C00h
int t2 = 0xdc0;
010516CC mov        dword ptr [t2],0DC0h
int t3 = 0;
010516D3 mov        dword ptr [t3],0

    t3 = s0 << 4;
010516DA mov        eax,dword ptr [s0]
010516DD shl        eax,4
010516E0 mov        dword ptr [t3],eax
int t0 = 0;
010516E3 mov        dword ptr [t0],0

    t0 = t1 & t2;
010516EA mov        eax,dword ptr [t1]
010516ED and        eax,dword ptr [t2]
010516F0 mov        dword ptr [t0],eax

    t0 = t1 | t2;
010516F3 mov        eax,dword ptr [t1]
010516F6 or         eax,dword ptr [t2]
010516F9 mov        dword ptr [t0],eax

    t0 = ~t1;
010516FC mov        eax,dword ptr [t1]
010516FF not        eax
01051701 mov        dword ptr [t0],eax
return 0;
01051704 xor        eax,eax
}

```

Figure 126: Disassembly code from visual studio that will run logical operations on bits of information on windows 32 bit compiler.

The screenshot shows the Visual Studio debugger interface with the following windows:

- Disassembly:** Shows the assembly code for the `main()` function. The assembly code includes instructions like `push ebp`, `mov esp,ebp`, `sub esp,0FCh`, etc., followed by a series of `int` and `mov` instructions. A specific instruction `t3 = s0 & t2;` is highlighted.
- Registers:** Shows the current register values. EAX contains `CCCCCC`, and EBP contains `00F9FBFC`.
- Memory:** Shows the memory dump starting at address `0x00F9FB94`. The memory is filled with the byte `CCCC` (hex `CCCC`) repeated across the range.
- Autos:** Shows the local variables `EAX` and `EBP` with their current values.
- Locals:** Not visible in the screenshot.
- Watch 1:** Not visible in the screenshot.
- Command Window:** Not visible in the screenshot.
- Immediate Window:** Not visible in the screenshot.
- Output:** Not visible in the screenshot.
- Error List:** Not visible in the screenshot.

Figure 127: Visual studio registers, memory and disassembly window after running one of the logical operations. As we can see the information stored to the right in memory is stored using big endian, the most significant byte is stored in the lowest addressable address.

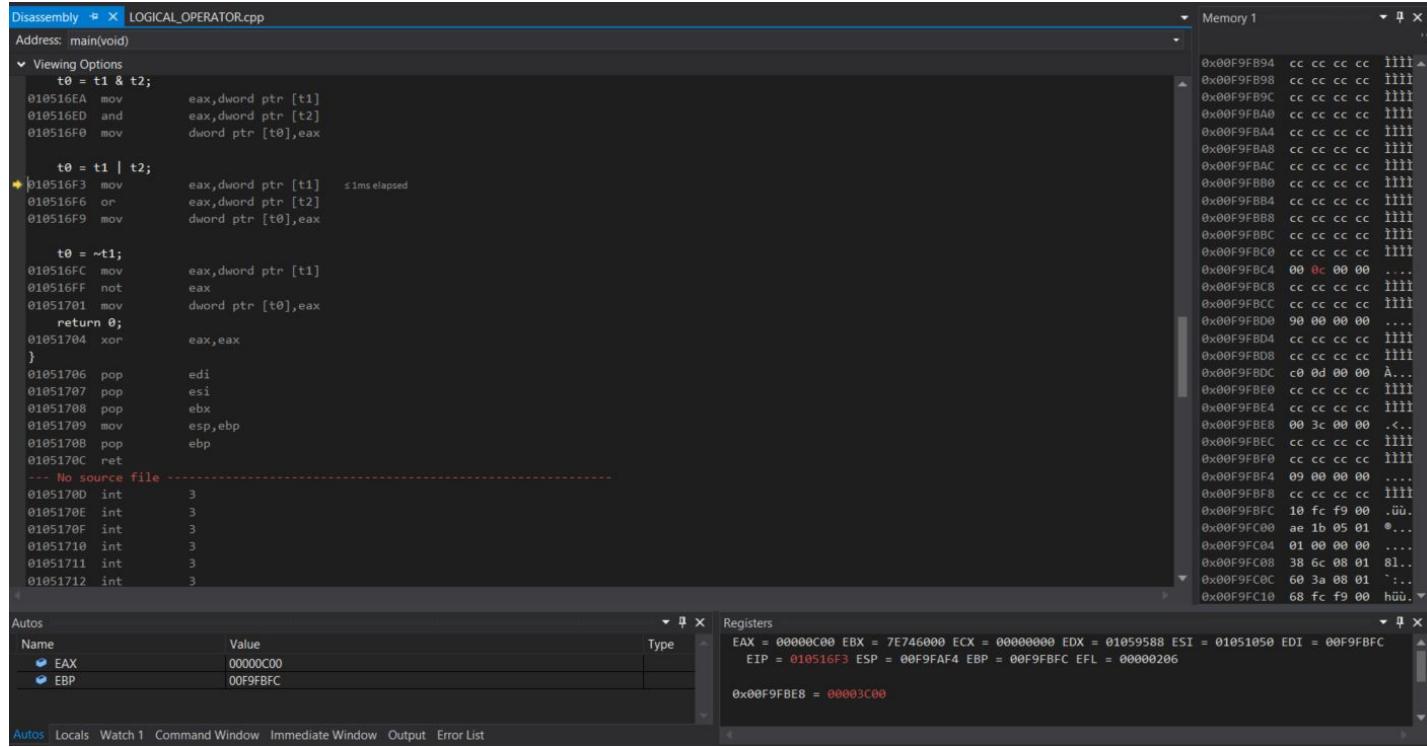
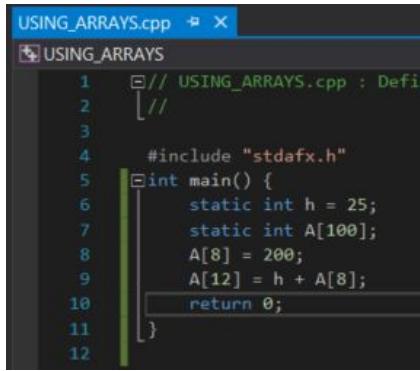


Figure 128: Visual studio showing in memory the information that is stored in reverse order than Linux 64 bit compiler or raspberry pi because we are using a windows 32 bit compiler and that uses big endian to store information. The operations are ran using the same method of moving information in memory and then manipulating it based on the desired logical operation that is going to be performed.

5.8 Using arrays on Windows 32 Bit Compiler

In this section we will display what happens in memory when we write to an address that is in an array even before the rest of the elements in the array are initialized.

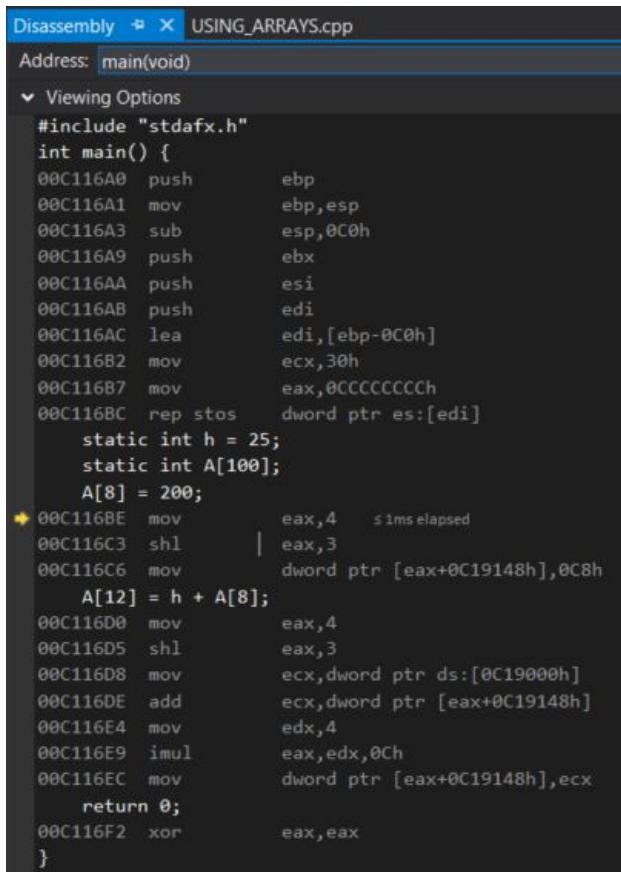


```

USING_ARRAYS.cpp ✘ X
USING_ARRAYS
1 //// USING_ARRAYS.cpp : Definition of the entry point for the application.
2 [//
3
4 #include "stdafx.h"
5 int main() {
6     static int h = 25;
7     static int A[100];
8     A[8] = 200;
9     A[12] = h + A[8];
10    return 0;
11 }
12

```

Figure 129: C++ code for using an array on windows 32 bit compiler.



Disassembly X USING_ARRAYS.cpp

Address: main(void)

Viewing Options

```

#include "stdafx.h"
int main() {
    push    ebp
    mov     ebp,esp
    sub    esp,0C0h
    push    ebx
    push    esi
    push    edi
    lea     edi,[ebp-0C0h]
    mov     ecx,30h
    mov     eax,0CCCCCCCCh
    rep stos dword ptr es:[edi]
    static int h = 25;
    static int A[100];
    A[8] = 200;
    mov     eax,4    $1ms elapsed
    shr     eax,3
    mov     dword ptr [eax+0C19148h],0C8h
    A[12] = h + A[8];
    mov     eax,4
    shr     eax,3
    mov     ecx,dword ptr ds:[0C19000h]
    add     ecx,dword ptr [eax+0C19148h]
    mov     edx,4
    imul   eax,edx,0Ch
    mov     dword ptr [eax+0C19148h],ecx
    return 0;
    xor     eax,eax
}

```

Figure 130: Disassembly code for using an array in windows 32 bit compiler. As we can see there is a large offset that is being used with respect to the eax in the third line and that is because it is setting the final location address from the array before it can be accessed.

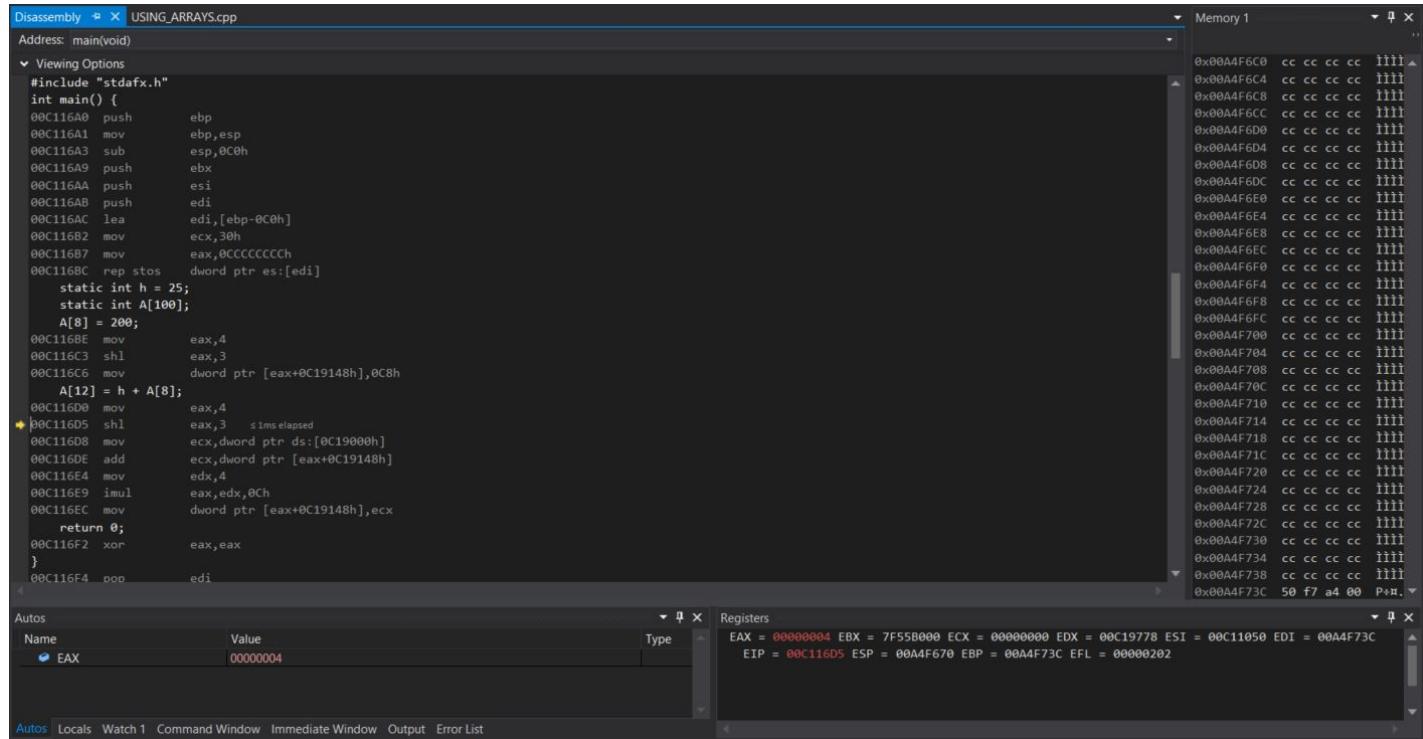


Figure 131: Visual studio assembly windows, registers and memory after the arrays value has been changed, adding the value of 25 to the value that is already 100.

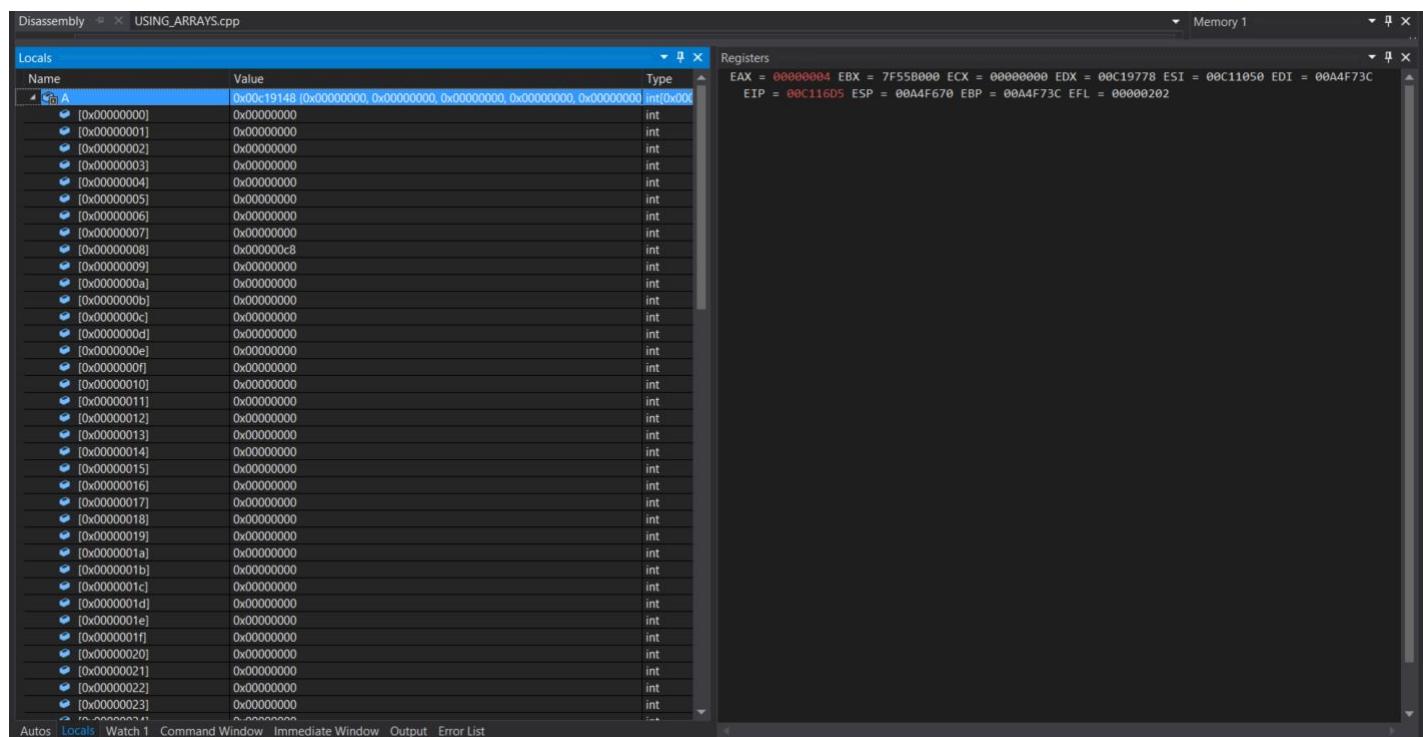


Figure 132: Locals window showing the contents of the array in memory for the windows 32 bit compiler that is using an array of 300 elements. In the address 0x00000008 we can see the value stored is 0x000000c8 or 225 as expected. Now we can see that the windows 32 bit compiler can also access the locations in an array without having to first initialized the rest of the elements in all of the other positions. Just like in linux 64 bit the raspberry pi and on MIPS.

6 Conclusion

Running these tests has made me better appreciate and understand that memory is very important in computers and that is why we have several different kinds of processors that handle memory differently. This is very important to programing because it helps understand how a computer is organized in a higher standard. I now understand that Linux and Mips use little endian and that windows 32 bit compiler uses big endian, otherwise if i would not have learned something like that in detail it could have been difficult to read information before first understanding what is going on in assembly in the background when I run a program. Now i am closer to understanding how a segmentation fault error occurs when for example i am trying to access an address that does not yet exist in memory or does not have usable information if i have not already defined access to that location. This must be how hackers study assembly in order to find exploits in programs just like for example each of the 4 compilers always make sure to initialize the base pointer and instruction pointer before doing any other operation, and deallocate it at the end. That can be exploited by a hacker especially considering that all compilers do that in order to create the stack frame for the variables and addresses and data that will be used and for the different instructions to know how to proceed in order when running a program.