Start time and date: 4/15/2022 10:00AM

End time and date: 4/15/2022 10:00PM

Christopher Lall

CSc 342/343 – Professor Gertner

Take Home Test 1

Due 4/17/222

I will neither give nor recieve unauthorized assistance on this TEST. I will only use one computing device to perform this TEST, I will not use cell while performing this TEST.

Chris L

Table of Contents

Objective: 4
Part 1: MIPS 4
2-2_1.asm 5
Registers: 9
2-2_2.asm 10
2-3_1.asm 13
2-3_2.asm: 17
2-5_2.asm: 21
2-6_1.asm: 24
While-Loop: 27
For Loop: 31
If-Then-Else: 35
Section 2.8: Write a simple “MAIN” in MIPS assembly that calls “myadd” function and analyze 39
Part 2: Intel x32 ISA in Microsoft Visual Studio 42
2-2_1.c 42
2-2_2.c 46
2-3_1.c 48
2-3_2.c 51
2-5_1.c 54
2-6_1.c 57
Natural_generator.c 61
While.c 63
For.c 66
Myadd.c 67
Part 3: Intel x86_64 ISA, Linux 64bit (GDB) 68
2-2_1.c 69
2-2_2.c 72
2-3_1 73
2-3_2.c 74
2-5_1.c 75
2-6_1.c 76
Natural_generator.c 77

Myadd.c 78
For.c 79
If.c 80
While.c 80
Comparison: 80
Mips: 80
Intel x32: 82
Linux: 82
Conclusion: 83

Objective:

The goal of this test is for students to demonstrate their understanding and ability to compare 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 with 64-bit gcc and gdb. Aside from that, this test uses the environment described above to assess students' grasp of the differences between local, static, and local static variables. We'll also look at how big endian and tiny endian are used in these two quite different settings.

Part 1: MIPS

The following pictures below for Part 1 will consist of the actual MIPS code and its disassembly, memory, and registers. I will thoroughly explain how each of the code MIPS code work individually, however I will go over the parts of a disassembly, memory, and registers windows using 2-2_1.asm as an example.

2-2_1.asm

2-2_1.asm

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

Figure 1. 2-2_1.asm file

Assembly in MARS MIPS is divided down into sections, which include .data and .text.

The Data Segment is where data is stored in memory (allocated in RAM), similar to variables in higher-level languages.

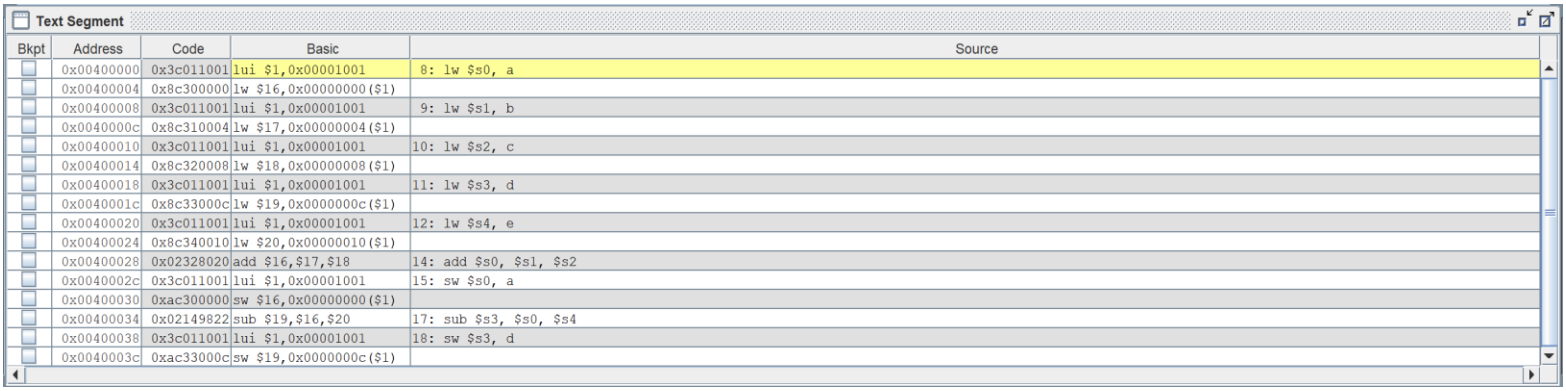
The Text Segment contains instructions and program logic

The .data section of MIPS is where static variables are declared. Static variables are data that the assembler allocates and whose size remains constant as the

program runs. Its contents change; "static" refers to the fact that the size in bytes does not vary throughout operation.

Local variables are declared in the function. After exiting that function, all local variables will be removed. Local variables, unlike static variables, do not require long-term storage, hence they are not specified in the .data section.

A local static variable is a special variable in which it is only initialized once no matter how many time the functions is called. It only lives within the scope of the function in which it is declared, but its lifetime begins when the function is called and ends only when the program is terminated.



Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001	lui $1,0x00001001	8: lw $s0, a
	0x00400004	0x8c300000	lw $16,0x00000000($1)	
	0x00400008	0x3c011001	lui $1,0x00001001	9: lw $s1, b
	0x0040000c	0x8c310004	lw $17,0x00000004($1)	
	0x00400010	0x3c011001	lui $1,0x00001001	10: lw $s2, c
	0x00400014	0x8c320008	lw $18,0x00000008($1)	
	0x00400018	0x3c011001	lui $1,0x00001001	11: lw $s3, d
	0x0040001c	0x8c33000c	lw $19,0x0000000c($1)	
	0x00400020	0x3c011001	lui $1,0x00001001	12: lw $s4, e
	0x00400024	0x8c340010	lw $20,0x00000010($1)	
	0x00400028	0x02328020	add $16,$17,$18	14: add $s0, $s1, $s2
	0x0040002c	0x3c011001	lui $1,0x00001001	15: sw $s0, a
	0x00400030	0xac300000	sw $16,0x00000000($1)	
	0x00400034	0x02149822	sub $19,$16,$20	17: sub $s3, $s0, $s4
	0x00400038	0x3c011001	lui $1,0x00001001	18: sw $s3, d
	0x0040003c	0xac33000c	sw $19,0x0000000c($1)	

Figure 2. MARS Simulator of disassembly window.

Bkpt – Stands for breakpoint. Indicates if a breakpoint is set at this address

Basic - Assembly language version (using core MIPS instruction set) of the instruction

Source: Actual source from your file (may differ from the Basic instruction if

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000001	0x00000002	0x00000003	0x00000004	0x00000005	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
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
0x100101e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Figure 3. Memory Window Mars simulator

source contains pseudo-instructions)

The first column in the memory pane provides the address that is used as a reference. The value of the address corresponding to the value inside the parenthesis is represented by the rest of the column, which is labeled value. The value of address 0x10010000 + 4 is shown in the column labeled "Value (+4)" or the one boxed in orange. In other words, the address 0x10010004 has the value 0x00000002.

Endian is the method of storing multiple-byte data in computers, such as short, int, long, float, and double. Data in MIPS is stored in a format known as "Big Endian." This signifies that the data's initial byte is saved first.

Registers | Coproc 1 | Coproc 0

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	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

Figure 4. Register Window in Mars Simulator

The picture on right is the registers window. I will go over every register.

Registers:

$zero : Zero register always hold the constant 0.

$at : Assembler temporary register reserved for assembler.

$v0 - $v1 : Return registers for returning values to caller function.

$a0 - $a3 : Argument registers for passing in arguments for subroutines.

$t0 - $t9 : Temporary registers used for intermediate calculations inside subroutines.

$s0 - $s7 : Saved registers where values are saved across subroutine calls.

$k0 - $k1 : Reserved for use by operating system and assembler.

$gp : Global pointer points to area in memory that saves global variables.

$sp : Stack Pointer which points to the last location in use on the stack.

$fp : Frame Pointer which points to the top of the stack.

$ra : Return address which is used by function call.

pc : Program counter which points to the next instruction to be executed.

hi, lo: They are not numbered registers, but they store the reuslts of operations that would not fit in a single register

2-2_2.asm

2-2_2.asm

```
1   .data
2   f: .word 0
3   g: .word 50
4   h: .word 40
5   i: .word 30
6   j: .word 20
7   .text
8   lw $s0, f
9   lw $s1, g
10  lw $s2, h
11  lw $s3, i
12  lw $s4, j
13  # t0 = g + h
14
15  add $t0, $s1, $s2
16  # t1 = i + j
17  add $t1, $s3, $s4
18  # f = t0 - t1
19  sub $s0, $t0, $t1
20  sw $s0, f
```

Figure 5. 2-2_2.asm code in Mars

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001	lui $1,0x00001001	8: lw $s0, f
	0x00400004	0x8c300000	lw $16,0x00000000($1)	
	0x00400008	0x3c011001	lui $1,0x00001001	9: lw $s1, g
	0x0040000c	0x8c310004	lw $17,0x00000004($1)	
	0x00400010	0x3c011001	lui $1,0x00001001	10: lw $s2, h
	0x00400014	0x8c320008	lw $18,0x00000008($1)	
	0x00400018	0x3c011001	lui $1,0x00001001	11: lw $s3, i
	0x0040001c	0x8c33000c	lw $19,0x0000000c($1)	
	0x00400020	0x3c011001	lui $1,0x00001001	12: lw $s4, j
	0x00400024	0x8c340010	lw $20,0x00000010($1)	
	0x00400028	0x02324020	add $8,$17,$18	15: add $t0, $s1, $s2
	0x0040002c	0x02744820	add $9,$19,$20	17: add $t1, $s3, $s4
	0x00400030	0x01098022	sub $16,$8,$9	19: sub $s0, $t0, $t1
	0x00400034	0x3c011001	lui $1,0x00001001	20: sw $s0, f
	0x00400038	0xac300000	sw $16,0x00000000($1)	

Figure 6. Text segment for 2-2_2.asm

Here, we are using the temporary registers in this code which is the difference

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000000	0x00000032	0x00000028	0x0000001e	0x00000014	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
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
0x100101e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Figure 7. Data segment for 2-2_2.asm

from part 2-2_1.

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x0000005a
$t1	9	0x00000032
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000028
$s1	17	0x00000032
$s2	18	0x00000028
$s3	19	0x0000001e
$s4	20	0x00000014
$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	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x0040003c
hi		0x00000000
lo		0x00000000

Figure 8. Registers segment for 2-2_2.asm

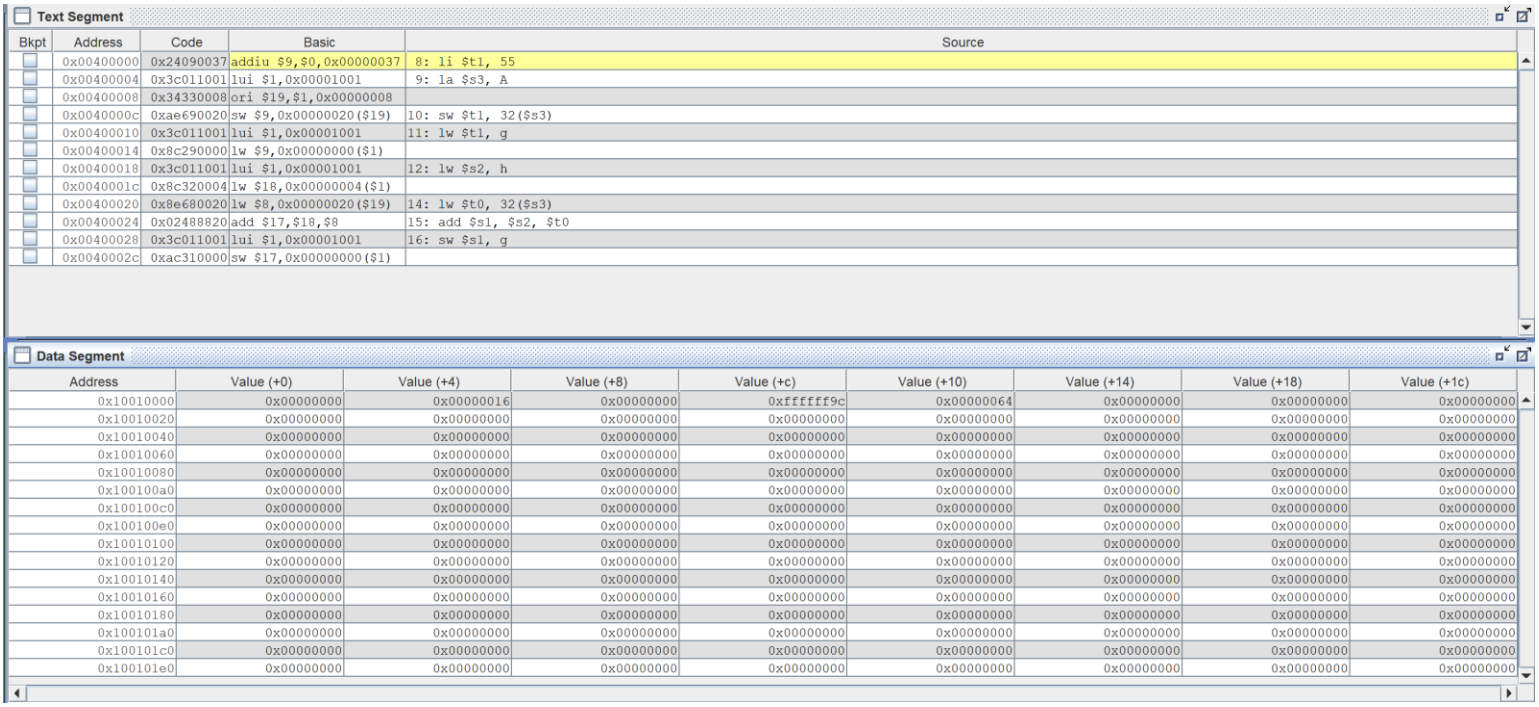
2-3_1.asm

2-3_1.asm

```
1   .data
2   g: .word 0
3   h: .word 22
4   A: .word 0-100
5   size : .word 100
6   .text
7   #just to set A[8] to 55
8   li $t1, 55
9   la $s3, A
10  sw $t1, 32($s3)
11  lw $t1, g
12  lw $s2, h
13  #loading the value of A[8] into t0
14  lw $t0, 32($s3)
15  add $s1, $s2, $t0
16  sw $s1, g
```

Figure 9. 2-3_1.asm code in Mars

We're using temporary registers and storing $s3 at a 32-bit offset. Because we are storing to A[8], and each hexadecimal value has four bytes, we must multiply 8 * 4 to get the offset value. This is not the beginning of array A.



Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x24090037	addiu $9,$0,0x00000037	8: li $t1, 55
	0x00400004	0x3c011001	lui $1,0x00001001	9: la $s3, A
	0x00400008	0x34330008	ori $19,$1,0x00000008	
	0x0040000c	0xae690020	sw $9,0x00000020($19)	10: sw $t1, 32($s3)
	0x00400010	0x3c011001	lui $1,0x00001001	11: lw $t1, g
	0x00400014	0x8c290000	lw $9,0x00000000($1)	
	0x00400018	0x3c011001	lui $1,0x00001001	12: lw $s2, h
	0x0040001c	0x8c320004	lw $18,0x00000004($1)	
	0x00400020	0x8e680020	lw $8,0x00000020($19)	14: lw $t0, 32($s3)
	0x00400024	0x02488820	add $17,$18,$8	15: add $s1, $s2, $t0
	0x00400028	0x3c011001	lui $1,0x00001001	16: sw $s1, g
	0x0040002c	0xac310000	sw $17,0x00000000($1)	

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000000	0x00000016	0x00000000	0xffffff9c	0x00000064	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
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
0x100101e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Figure 10. Text and data segment for 2-3_1.asm co

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000037
$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	0x0000004d
$s2	18	0x00000016
$s3	19	0x10010008
$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	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400030
hi		0x00000000
lo		0x00000000

Figure 11. Registers portion for 2-3_1.asm

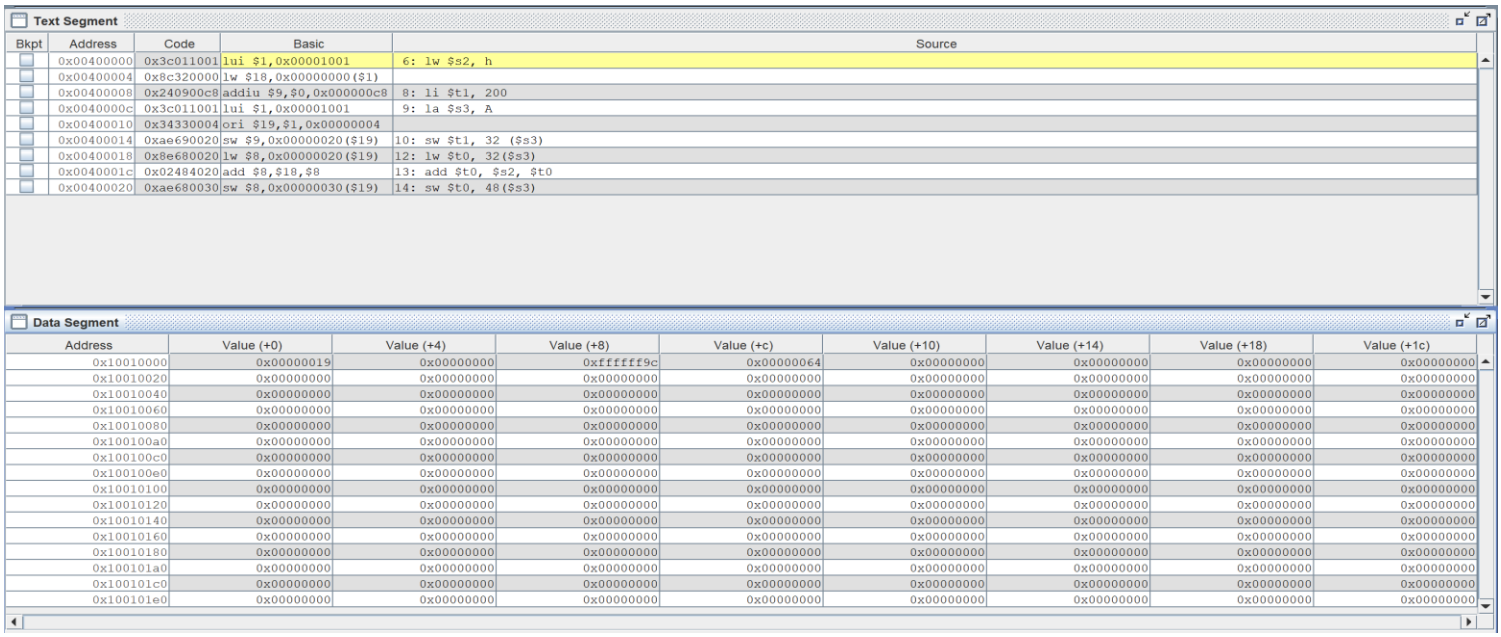
2-3_2.asm:

Edit Execute

2-3_2.asm

```
1   .data
2   h: .word 25
3   A: .word 0-100
4   size: .word 100
5   .text
6   lw $s2, h
7   #initializing A[8] to 200
8   li $t1, 200
9   la $s3, A
10  sw $t1, 32 ($s3)
11  #A[12] = h + A[8]
12  lw $t0, 32($s3)
13  add $t0, $s2, $t0
14  sw $t0, 48($s3)
15
```

Figure 12. 2-3_2.asm code



Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001	lui $1,0x00001001	6: lw $s2, h
	0x00400004	0x8c320000	lw $18,0x00000000($1)	
	0x00400008	0x240900c8	addiu $9,$0,0x000000c8	8: li $t1, 200
	0x0040000c	0x3c011001	lui $1,0x00001001	9: la $s3, A
	0x00400010	0x34330004	ori $19,$1,0x00000004	
	0x00400014	0xae690020	sw $9,0x00000020($19)	10: sw $t1, 32 ($s3)
	0x00400018	0x8e680020	lw $8,0x00000020($19)	12: lw $t0, 32($s3)
	0x0040001c	0x02484020	add $8,$18,$8	13: add $t0, $s2, $t0
	0x00400020	0xae680030	sw $8,0x00000030($19)	14: sw $t0, 48($s3)

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000019	0x00000000	0xffffff9c	0x00000064	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
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
0x100101e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Figure 13. Text and data segment for 2-3_2.asm code

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x000000e1
$t1	9	0x000000c8
$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	0x00000019
$s3	19	0x10010004
$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	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400024
hi		0x00000000
lo		0x00000000

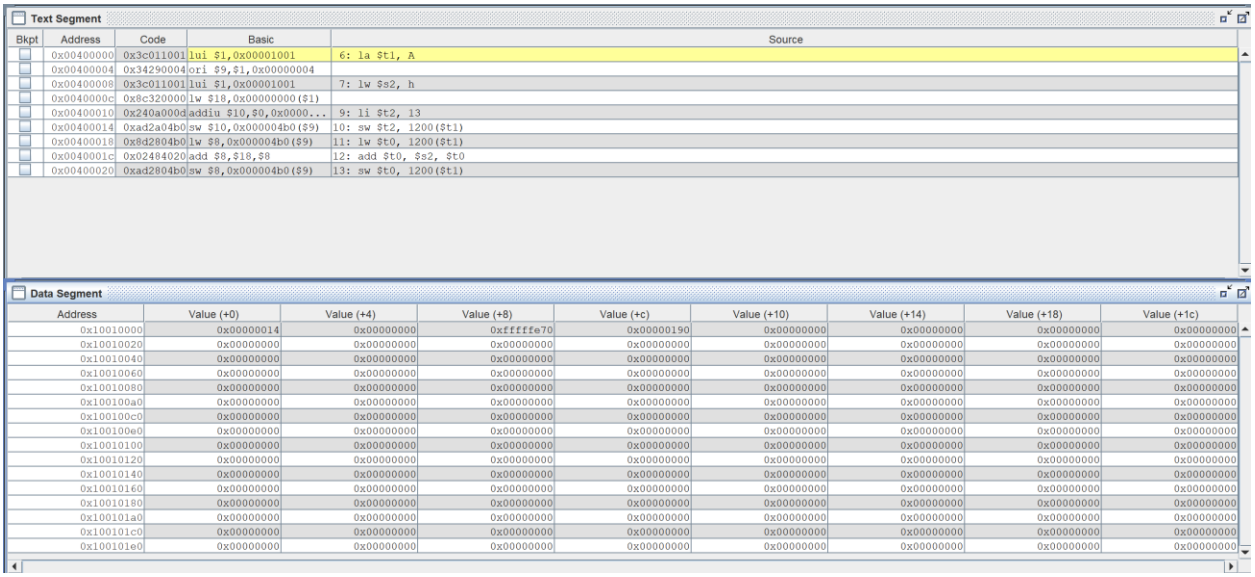
Figure 14. Registers for 2-3_2.asm code

2-5_2.asm:

2-5_2.asm

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

Figure 15. 2-5_2.asm code in Mars



Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001	lui $1,0x00001001	6: la $t1, A
	0x00400004	0x34290004	ori $9,$1,0x00000004	
	0x00400008	0x3c011001	lui $1,0x00001001	7: lw $s2, h
	0x0040000c	0x8c320000	lw $18,0x00000000($1)	
	0x00400010	0x240a000d	addiu $10,$0,0x0000...	9: li $t2, 13
	0x00400014	0xad2a04b0	sw $10,0x000004b0($9)	10: sw $t2, 1200($t1)
	0x00400018	0x8d2804b0	lw $8,0x000004b0($9)	11: lw $t0, 1200($t1)
	0x0040001c	0x02484020	add $8,$18,$8	12: add $t0, $s2, $t0
	0x00400020	0xad2804b0	sw $8,0x000004b0($9)	13: sw $t0, 1200($t1)

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000014	0x00000000	0xfffffe70	0x00000190	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
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
0x100101e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Figure 16. Text and data segment for 2-5_2.asm code

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000021
$t1	9	0x10010004
$t2	10	0x0000000d
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000014
$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	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400024
hi		0x00000000
lo		0x00000000

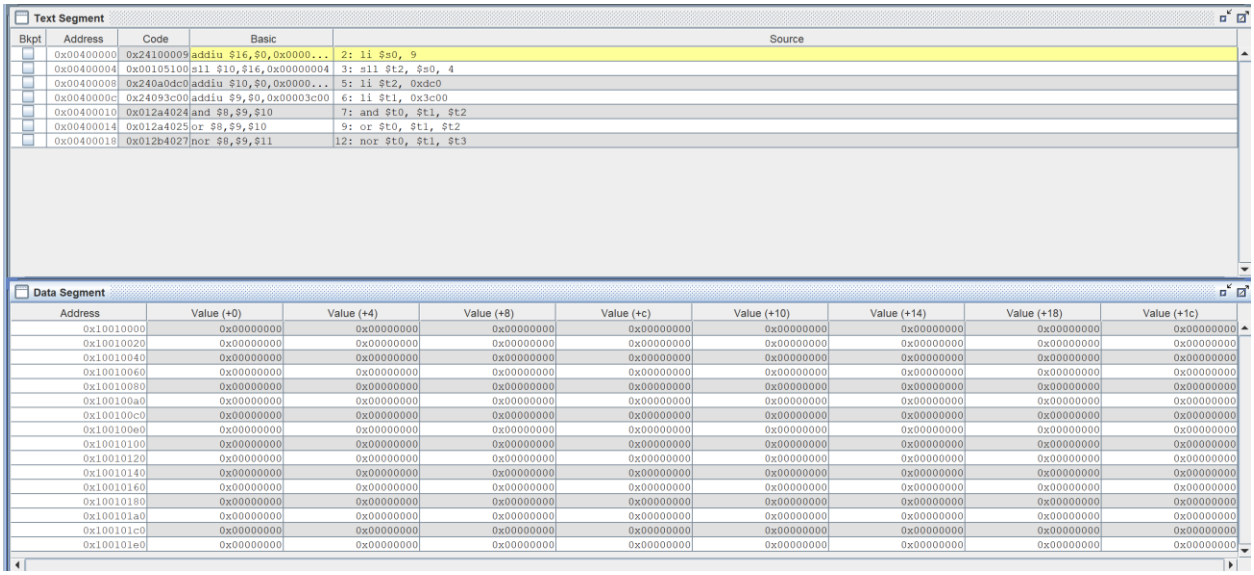
Figure 17. Registers for 2-5_2.asm

2-6_1.asm:

2-6_1.asm

```
1   #left shift
2   li $s0, 9
3   sll $t2, $s0, 4
4   # AND
5   li $t2, 0xdc0
6   li $t1, 0x3c00
7   and $t0, $t1, $t2
8   # OR
9   or $t0, $t1, $t2
10  # NOR li
11  #t3, 0
12  nor $t0, $t1, $t3
```

Figure 18. 2-6_1.asm code in Mars



Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x24100009	addiu $16,$0,0x0000...	2: li $s0, 9
	0x00400004	0x00105100	sll $10,$16,0x00000004	3: sll $t2, $s0, 4
	0x00400008	0x240a0dc0	addiu $10,$0,0x0000...	5: li $t2, 0xdc0
	0x0040000c	0x24093c00	addiu $9,$0,0x00003c00	6: li $t1, 0x3c00
	0x00400010	0x012a4024	and $8,$9,$10	7: and $t0, $t1, $t2
	0x00400014	0x012a4025	or $8,$9,$10	9: or $t0, $t1, $t2
	0x00400018	0x012b4027	nor $8,$9,$11	12: nor $t0, $t1, $t3

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
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
0x100101e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Figure 19. Text and Data segment for 2-6_1.asm code

Registers | Coproc 1 | Coproc 0

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	0xffffc3ff
$t1	9	0x00003c00
$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	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x0040001c
hi		0x00000000
lo		0x00000000

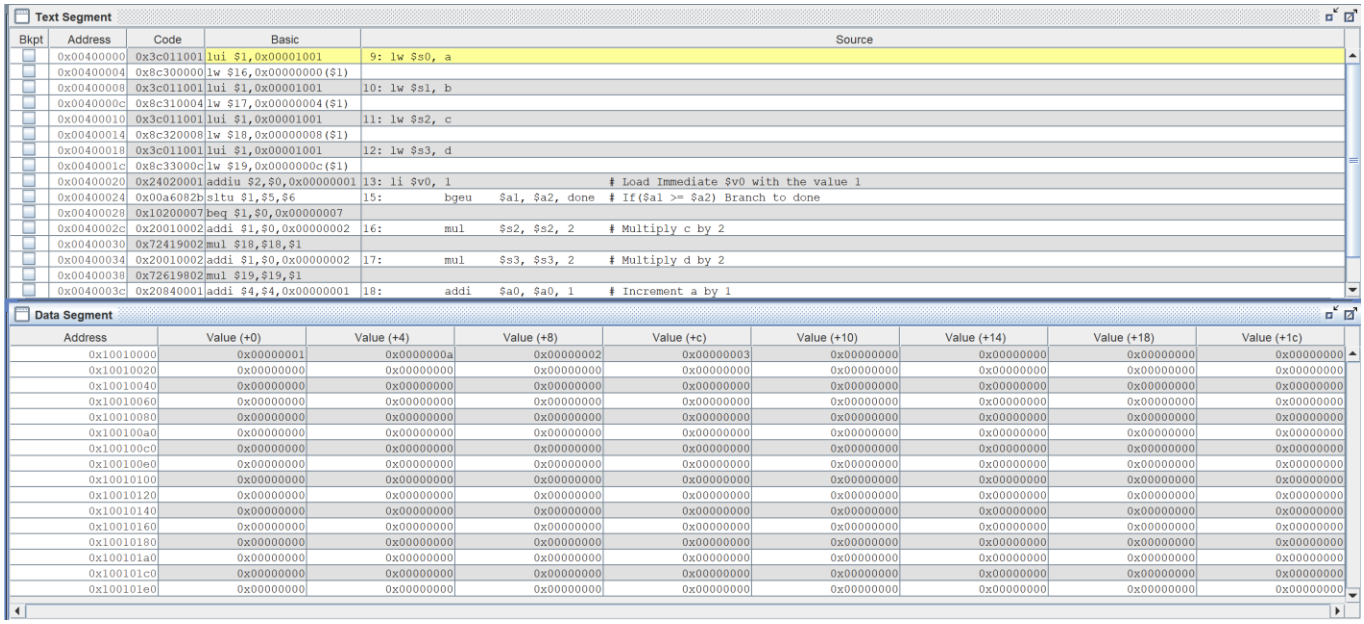
Figure 20. Register for 2-6_1.asm code

While-Loop:

whileLoop.asm

```
1  .data #Initialize a to d their respective values
2  a: .word 1
3  b: .word 10
4  c: .word 2
5  d: .word 3
6
7  .text
8  # Copy values from memory
9  lw $s0, a
10 lw $s1, b
11 lw $s2, c
12 lw $s3, d
13 li $v0, 1                        # Load Immediate $v0 with the value 1
14 loop:
15         bgeu    $a1, $a2, done  # If($a1 >= $a2) Branch to done
16         mul     $s2, $s2, 2     # Multiply c by 2
17         mul     $s3, $s3, 2     # Multiply d by 2
18         addi    $a0, $a0, 1     # Increment a by 1
19         addi    $a2, $a2, -1    # Decrement b by 1
20         b       loop            # Branch to loop
21
22 done:
23
```

Figure 21. whileLoop.asm code in Mars



Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001	lui $1,0x00001001	9: lw $s0, a
	0x00400004	0x8c300000	lw $16,0x00000000($1)	
	0x00400008	0x3c011001	lui $1,0x00001001	10: lw $s1, b
	0x0040000c	0x8c310004	lw $17,0x00000004($1)	
	0x00400010	0x3c011001	lui $1,0x00001001	11: lw $s2, c
	0x00400014	0x8c320008	lw $18,0x00000008($1)	
	0x00400018	0x3c011001	lui $1,0x00001001	12: lw $s3, d
	0x0040001c	0x8c33000c	lw $19,0x0000000c($1)	
	0x00400020	0x24020001	addiu $2,$0,0x00000001	13: li $v0, 1 # Load Immediate $v0 with the value 1
	0x00400024	0x00a6082b	sltu $1,$5,$6	15: bgeu $a1, $a2, done # If($a1 >= $a2) Branch to done
	0x00400028	0x10200007	beq $1,$0,0x00000007	
	0x0040002c	0x20010002	addi $1,$0,0x00000002	16: mul $s2, $s2, 2 # Multiply c by 2
	0x00400030	0x72419002	mul $18,$18,$1	
	0x00400034	0x20010002	addi $1,$0,0x00000002	17: mul $s3, $s3, 2 # Multiply d by 2
	0x00400038	0x72619802	mul $19,$19,$1	
	0x0040003c	0x20840001	addi $4,$4,0x00000001	18: addi $a0, $a0, 1 # Increment a by 1

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000001	0x0000000a	0x00000002	0x00000003	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
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
0x100101e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Figure 22. Text and Data segment for whileLoop.asm

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000001
$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	0x00000001
$s1	17	0x0000000a
$s2	18	0x00000002
$s3	19	0x00000003
$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	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400048
hi		0x00000000
lo		0x00000000

Figure 23. Registers for whileLoop.asm

The following is an example of MIPS using while statement. This MIPS code is equivalent to the C code shown below:

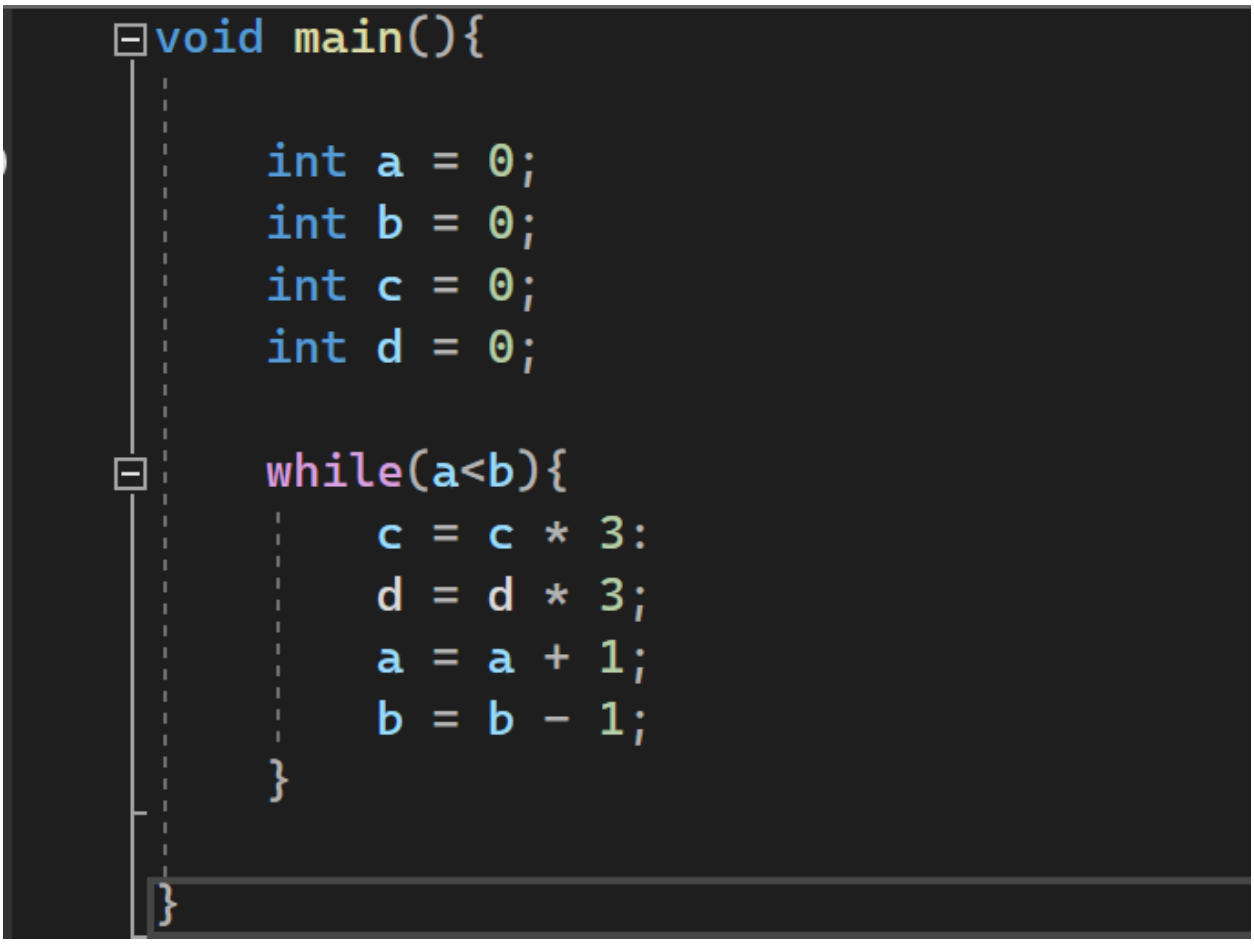


Figure 24. while loop .c file equivalent to mips

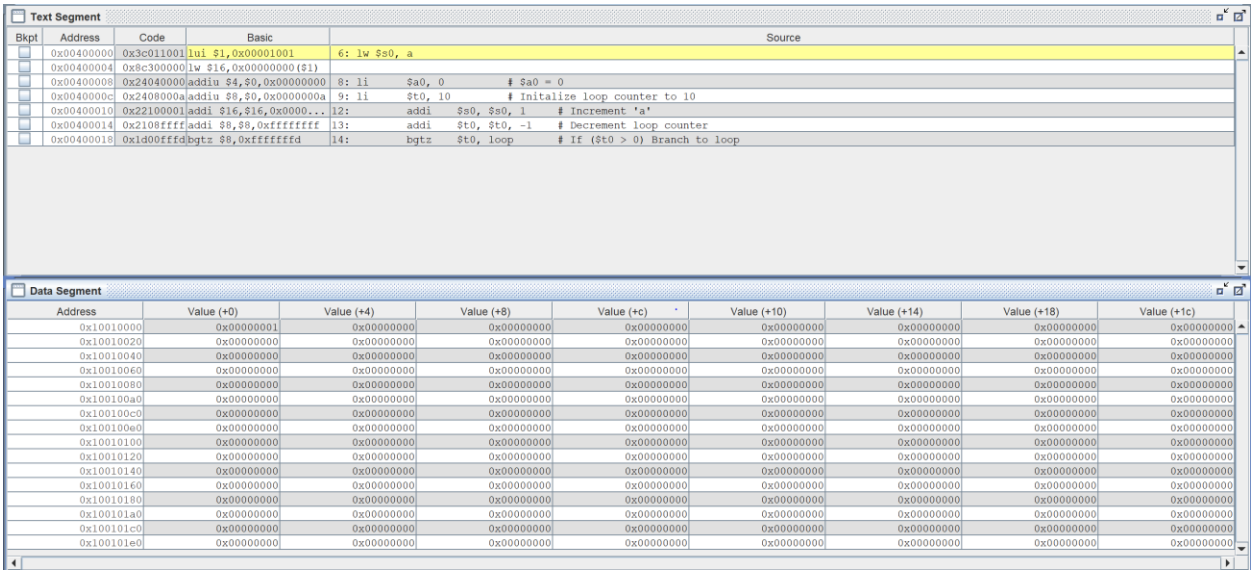
For Loop:

Edit Execute

forloop.asm

```
1   .data #Initialize a to d their respective values
2   a: .word 1
3
4   .text
5   # Copy values from memory
6   lw $s0, a
7
8   li      $a0, 0          # $a0 = 0
9   li      $t0, 10         # Initalize loop counter to 10
10
11  loop:
12          addi    $s0, $s0, 1     # Increment 'a'
13          addi    $t0, $t0, -1    # Decrement loop counter
14          bgtz    $t0, loop       # If ($t0 > 0) Branch to loop
15
```

Figure 25. For loop code in Mars



Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001	lui $1,0x00001001	6: lw $s0, a
	0x00400004	0x8c300000	lw $16,0x00000000($1)	
	0x00400008	0x24040000	addiu $4,$0,0x00000000	8: li $a0, 0 # $a0 = 0
	0x0040000c	0x2408000a	addiu $8,$0,0x0000000a	9: li $t0, 10 # Initalize loop counter to 10
	0x00400010	0x22100001	addi $16,$16,0x0000...	12: addi $s0, $s0, 1 # Increment 'a'
	0x00400014	0x2108ffff	addi $8,$8,0xffffffff	13: addi $t0, $t0, -1 # Decrement loop counter
	0x00400018	0x1d00fffd	bgtz $8,0xfffffffd	14: bgtz $t0, loop # If ($t0 > 0) Branch to loop

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000001	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
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
0x100101e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Figure 26. Text and Data segment for forloop code

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$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	0x0000000b
$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	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x0040001c
hi		0x00000000
lo		0x00000000

Figure 27. Registers for forloop code

The following is an example of MIPS using a for-loop statement. This MIPS code is equivalent to the C code shown below:

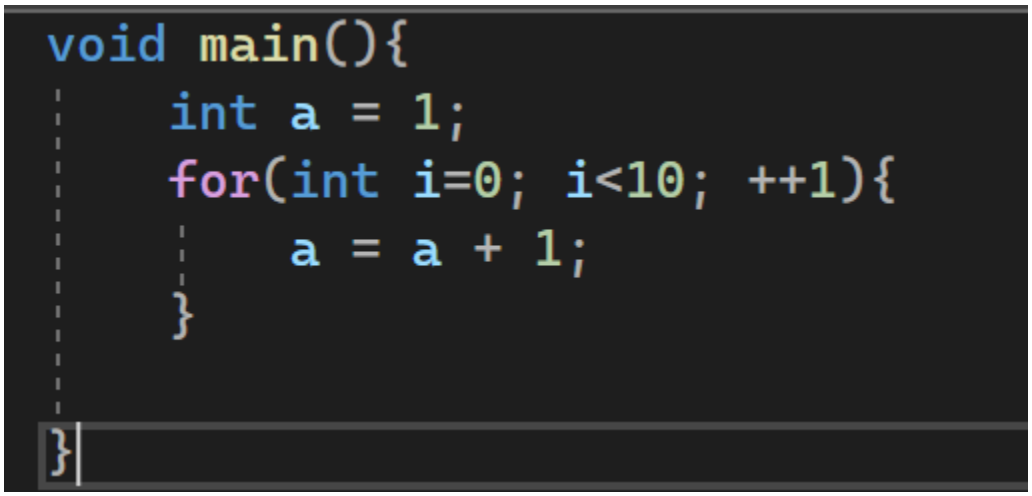


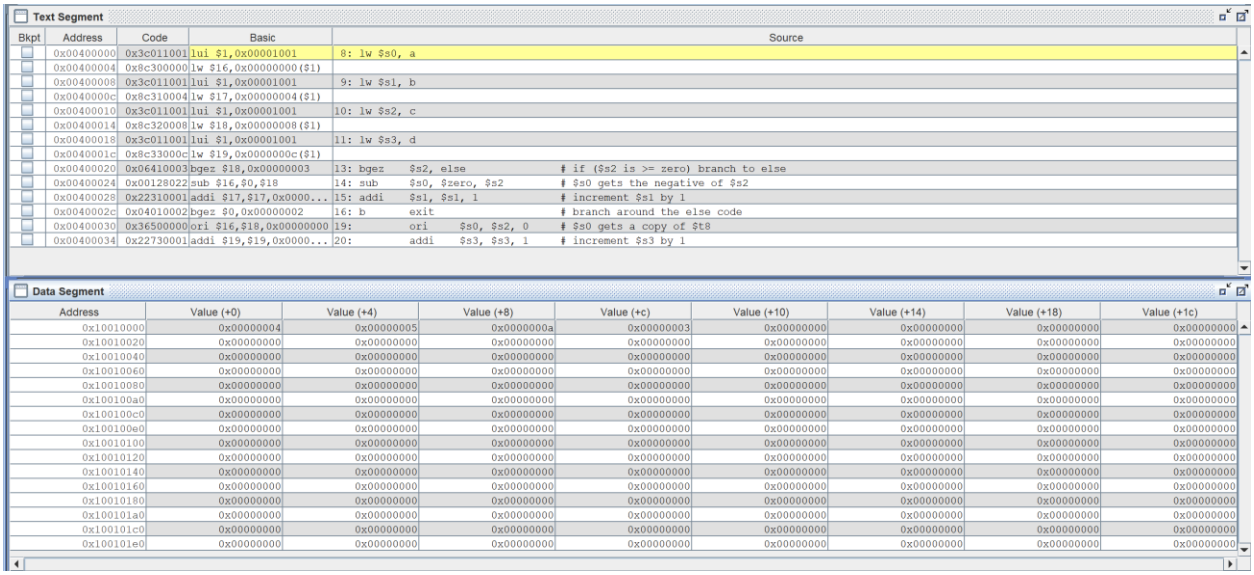
Figure 28. forloop.c equivalent of mips file

If-Then-Else:

ifThenElse.asm

```
1  .data #Initialize a to d their respective values
2  a: .word 4
3  b: .word 5
4  c: .word 10
5  d: .word 3
6  .text
7  # Copy values from memory
8  lw $s0, a
9  lw $s1, b
10 lw $s2, c
11 lw $s3, d
12 #if-else statement
13 bgez    $s2, else              # if ($s2 is >= zero) branch to else
14 sub     $s0, $zero, $s2        # $s0 gets the negative of $s2
15 addi    $s1, $s1, 1            # increment $s1 by 1
16 b       exit                   # branch around the else code
17
18 else:
19         ori     $s0, $s2, 0    # $s0 gets a copy of $t8
20         addi    $s3, $s3, 1    # increment $s3 by 1
21 exit:
```

Figure 29. If-then-else.asm code in Mars



Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001	lui $1,0x00001001	8: lw $s0, a
	0x00400004	0x8c300000	lw $16,0x00000000($1)	
	0x00400008	0x3c011001	lui $1,0x00001001	9: lw $s1, b
	0x0040000c	0x8c310004	lw $17,0x00000004($1)	
	0x00400010	0x3c011001	lui $1,0x00001001	10: lw $s2, c
	0x00400014	0x8c320008	lw $18,0x00000008($1)	
	0x00400018	0x3c011001	lui $1,0x00001001	11: lw $s3, d
	0x0040001c	0x8c33000c	lw $19,0x0000000c($1)	
	0x00400020	0x06410003	bgez $18,0x00000003	13: bgez $s2, else # if ($s2 is >= zero) branch to else
	0x00400024	0x00128022	sub $16,$0,$18	14: sub $s0, $zero, $s2 # $s0 gets the negative of $s2
	0x00400028	0x22310001	addi $17,$17,0x0000...	15: addi $s1, $s1, 1 # increment $s1 by 1
	0x0040002c	0x04010002	bgez $0,0x00000002	16: b exit # branch around the else code
	0x00400030	0x36500000	ori $16,$18,0x00000000	19: ori $s0, $s2, 0 # $s0 gets a copy of $t8
	0x00400034	0x22730001	addi $19,$19,0x0000...	20: addi $s3, $s3, 1 # increment $s3 by 1

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000004	0x00000005	0x0000000a	0x00000003	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
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
0x100101e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

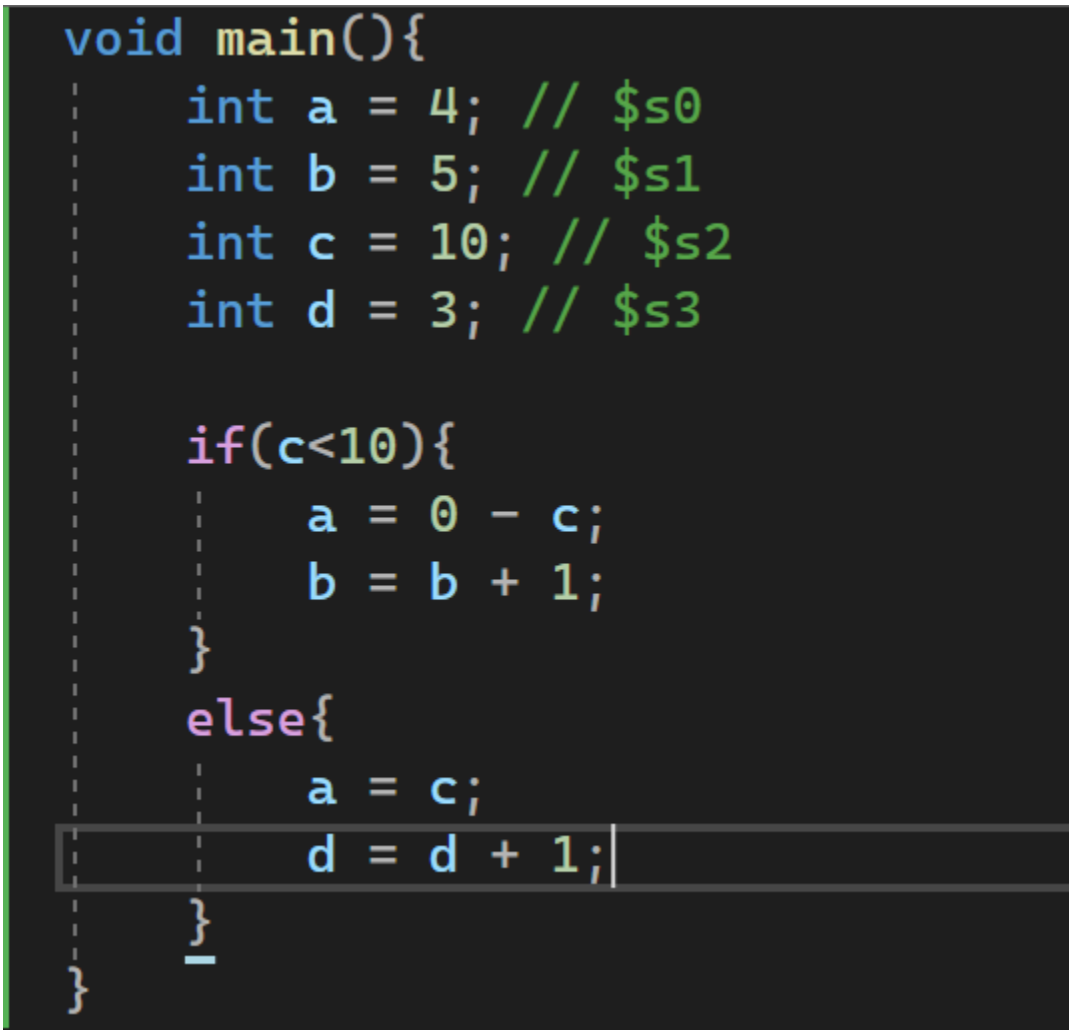
Figure 30. Text and Data segment for if-then-else.asm code

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$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	0x0000000a
$s1	17	0x00000005
$s2	18	0x0000000a
$s3	19	0x00000004
$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	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400038
hi		0x00000000
lo		0x00000000

Figure 31. Registers for if-then-else code

The following is an example of MIPS using an if-else statement. This MIPS code is equivalent to the C code shown below:



```c
void main(){
    int a = 4; // $s0
    int b = 5; // $s1
    int c = 10; // $s2
    int d = 3; // $s3

    if(c<10){
        a = 0 - c;
        b = b + 1;
    }
    else{
        a = c;
        d = d + 1;
    }
}
```

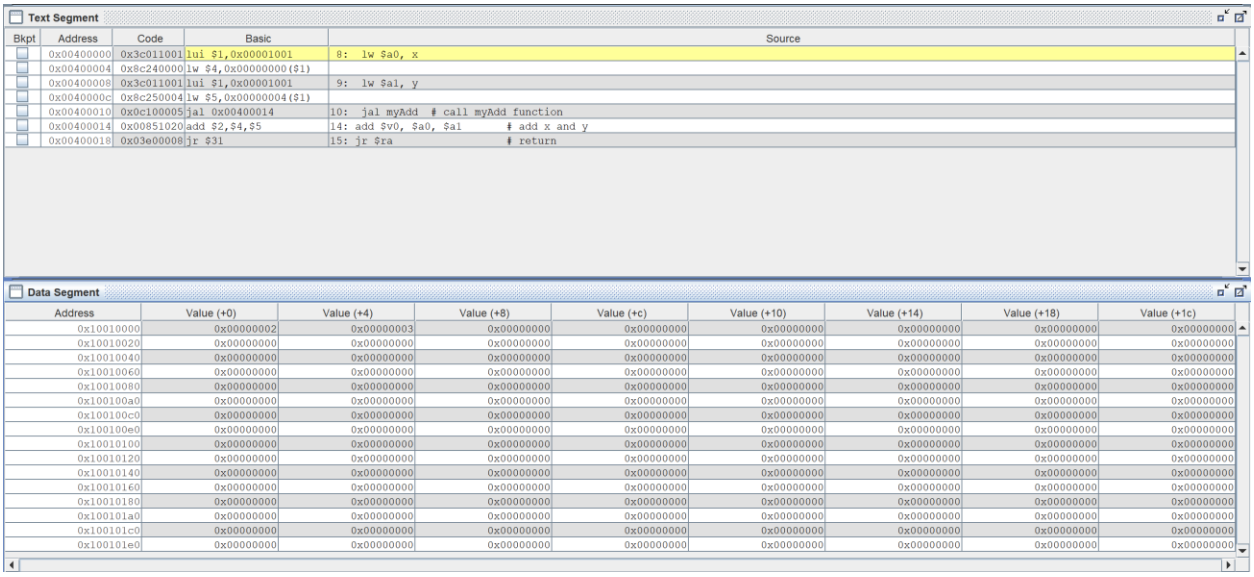
Figure 32. ifThenElse.c equivalent to mips

Section 2.8: Write a simple "MAIN" in MIPS assembly that calls "myadd" function and analyze

myadd.asm

```
1  .data            #Initialize a to d their respective values
2  x: .word 2
3  y: .word 3
4
5  .text
6  # saving x and y values into argument registers
7  # Use li to pass in parameter to argument register
8   lw $a0, x
9   lw $a1, y
10  jal myAdd  # call myAdd function
11
12 myAdd:
13 # function with 2 arguments and load value in return register
14 add $v0, $a0, $a1        # add x and y
15 jr $ra                   # return
16
```

Figure 33. myadd.asm in MARS



Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001	lui $1,0x00001001	8: lw $a0, x
	0x00400004	0x8c240000	lw $4,0x00000000($1)	
	0x00400008	0x3c011001	lui $1,0x00001001	9: lw $a1, y
	0x0040000c	0x8c250004	lw $5,0x00000004($1)	
	0x00400010	0x0c100005	jal 0x00400014	10: jal myAdd # call myAdd function
	0x00400014	0x00851020	add $2,$4,$5	14: add $v0, $a0, $a1 # add x and y
	0x00400018	0x03e00008	jr $31	15: jr $ra # return

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000002	0x00000003	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
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
0x100101e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Figure 34. Text and Data segment for myadd code

Registers | Coproc 1 | Coproc 0

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	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

Figure 35. Registers for myadd.asm

Part 2: Intel x32 ISA in Microsoft Visual Studio

The actual C code, as well as its disassembly, memory, and registers, will be shown in the accompanying images for Part 2. I'll go over the parts of a disassembly, memory, and registers windows utilizing 2- 2_1, but I'll go over the parts of a disassembly, memory, and registers windows separately. Consider the following example.

2-2_1.c

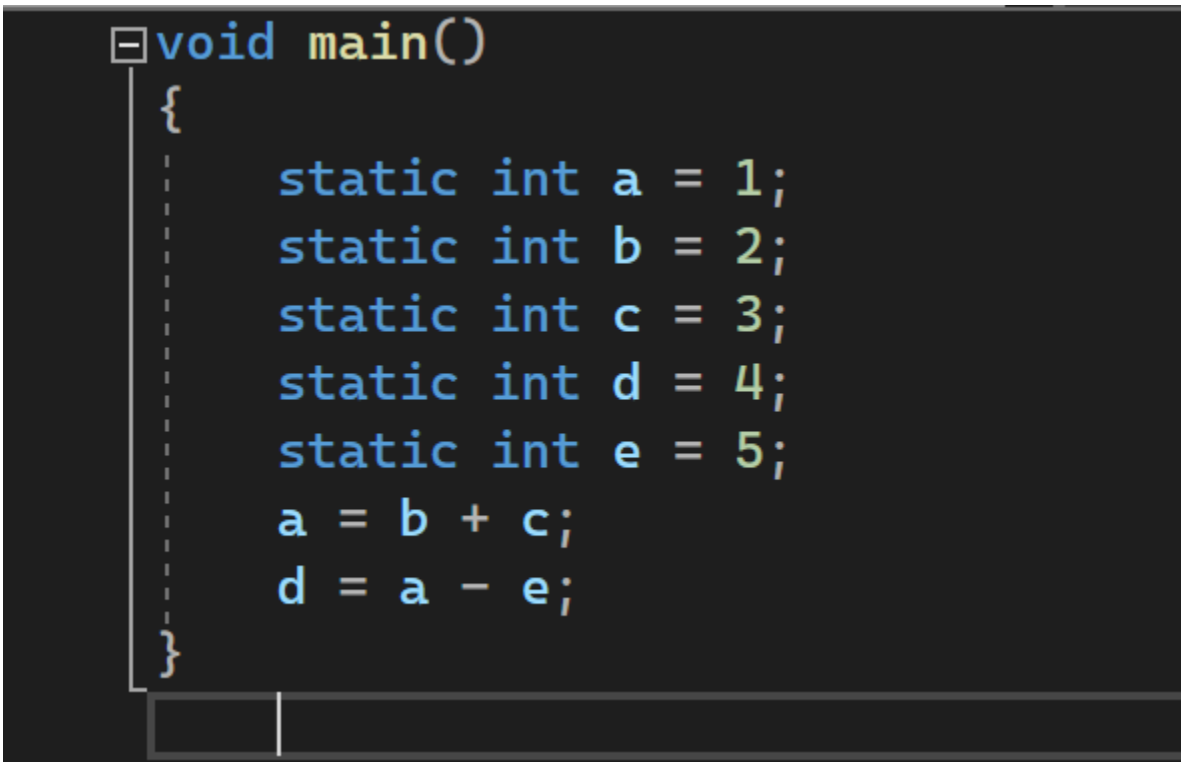


Figure 36. 2-2_1.c

It's pretty easy to tell what's local, static, and local static variables in Intel X32. If a variable has the static keyword before the int, it is considered static. A local variable is defined within a function, including the main function, and does not have the term static.

A static local variable, on the other hand, would have the keyword static, but it would be defined in a user-defined function rather than the main function. Because it is defined in the main function, the image above illustrates a simple static variable.

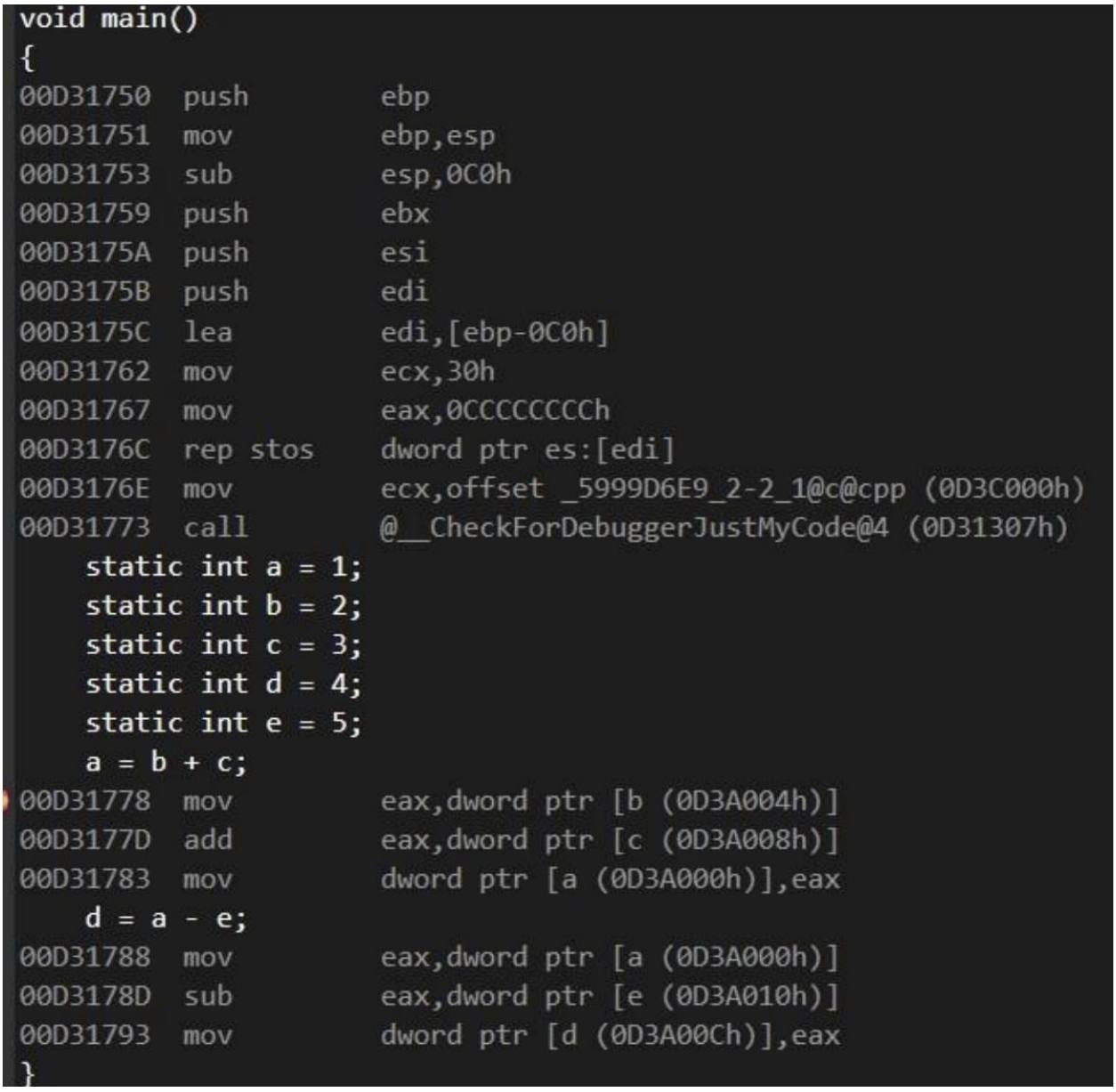


Figure 37. 2-2_1.c disassembly

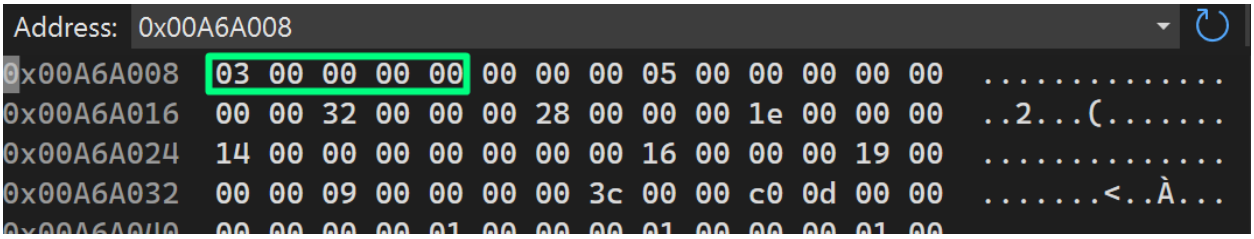


Figure 38. 2-2_1.c memory window VS

The above picture is the memory window.

Endian is the method of storing multiple-byte data in computers, such as short, int, long, float, and double. Data in Intel X32 is stored in a format known as "Little Endian." This signifies that the data's initial byte is saved last. Consider the address 0x00A6A008, which is presented in the first row of the memory pane. "03 00 00 00" is the value assigned to that address. The most significant bits are kept last since it is stored in little endian. As a result, the address's true value is "00 00 00 03." In general, Little Endian is preferable to Big Endian. The main advantage of Little Endian is that the value in a range of types is easy to read.

Figure 39. register window

There are 10 registers in Intel X32 as shown in the register window. I will go over all these registers.

EAX : Accumulator register which is used for I/O port access, arithmetic, interrupt calls, etc.

ECX : Counter register which is used as a loop counter and for shifts.

ESI / EDI : General purpose registers that can only add register values

EIP : Instruction pointer that points to the first byte of the next instruction to be executed.

EBP : Frame pointer that contains the base address of the functions' frame

EBX : Base register, which is used for memory access

EDX : Data register, which is used for I/O port access, arithmetic, some interrupt calls

ESP : Stack pointer which means it holds the top of the stack.

EFL : Short for EFLAGS, contains a set of flags that control the operation of the CPU

2-2_2.c

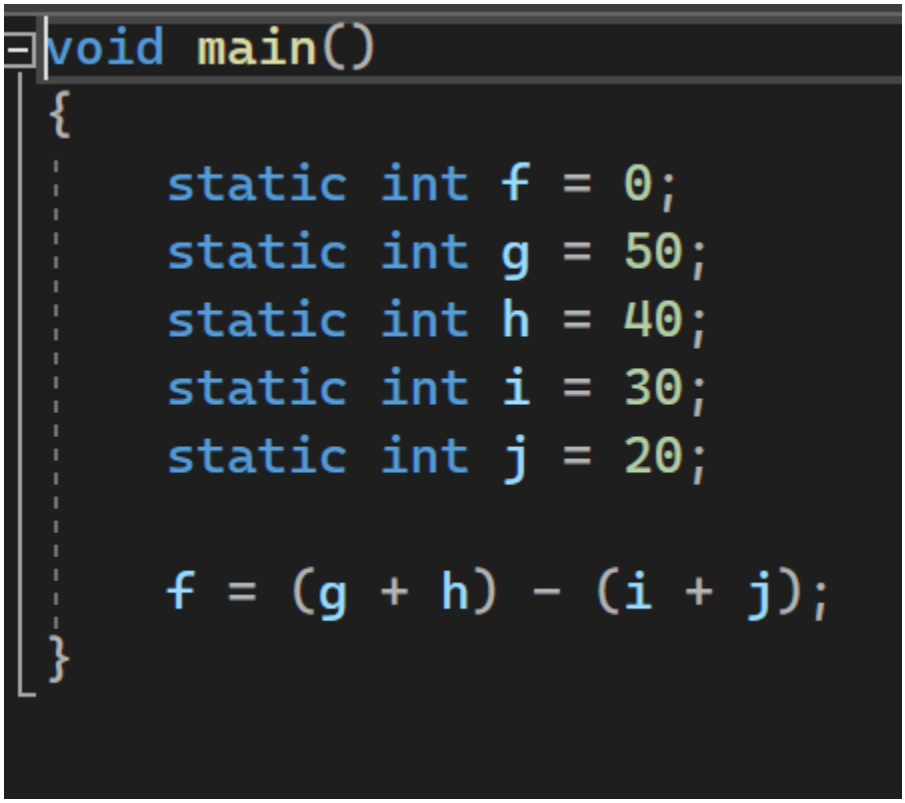


Figure 40. 2-2_2.c in VS

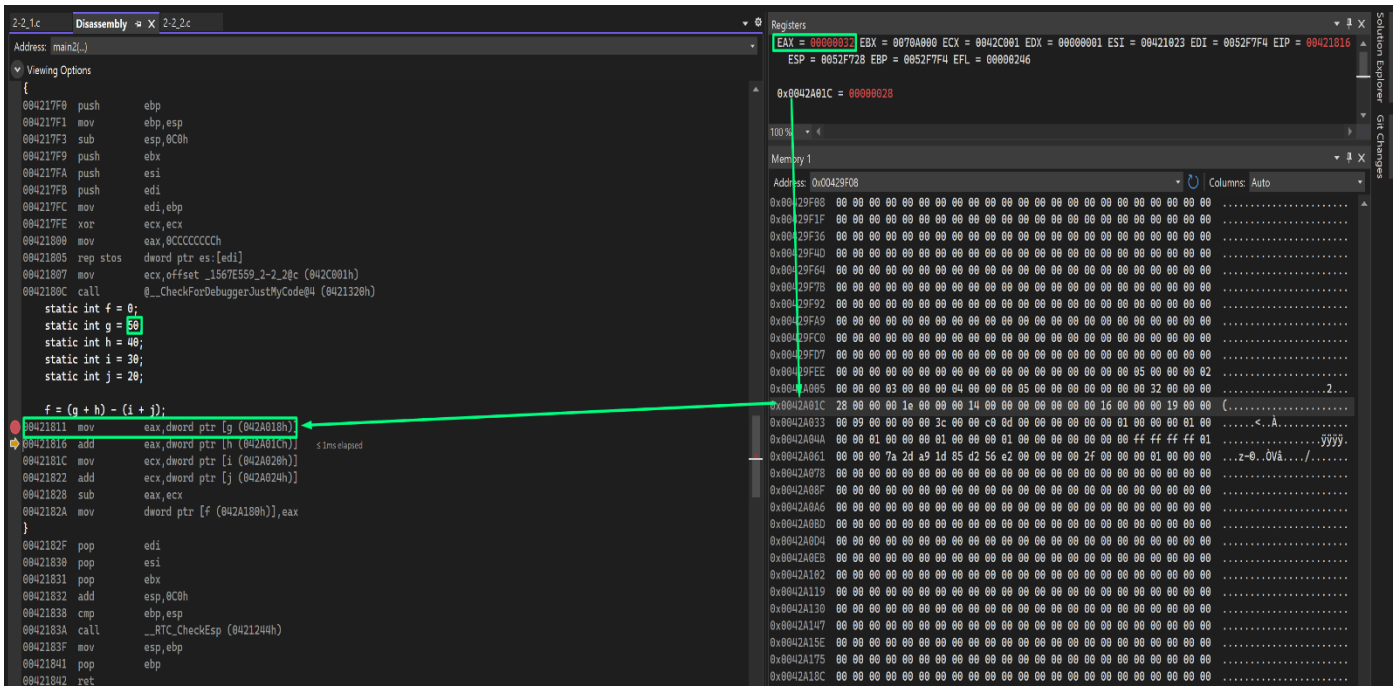


Figure 41. variable g being stored.

In figure 41, we see that g is stored in EAX register then into memory.

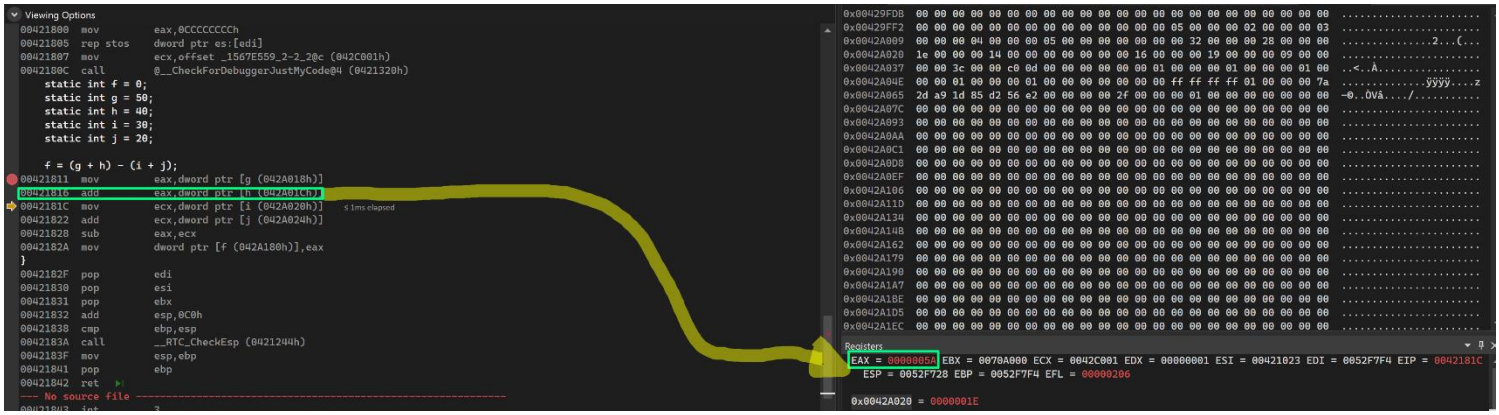


Figure 42. adding g+h.

In this figure we see that g and h are being added and then the result is stored in EAX and in the memory.

2-3_1.c

```
void main3()
{
    static int g = 0;
    static int h = 22;
    static int A[100];
    A[8] = 55;
    g = h + A[8];
}
```

Figure 43. 2-3_1.c code

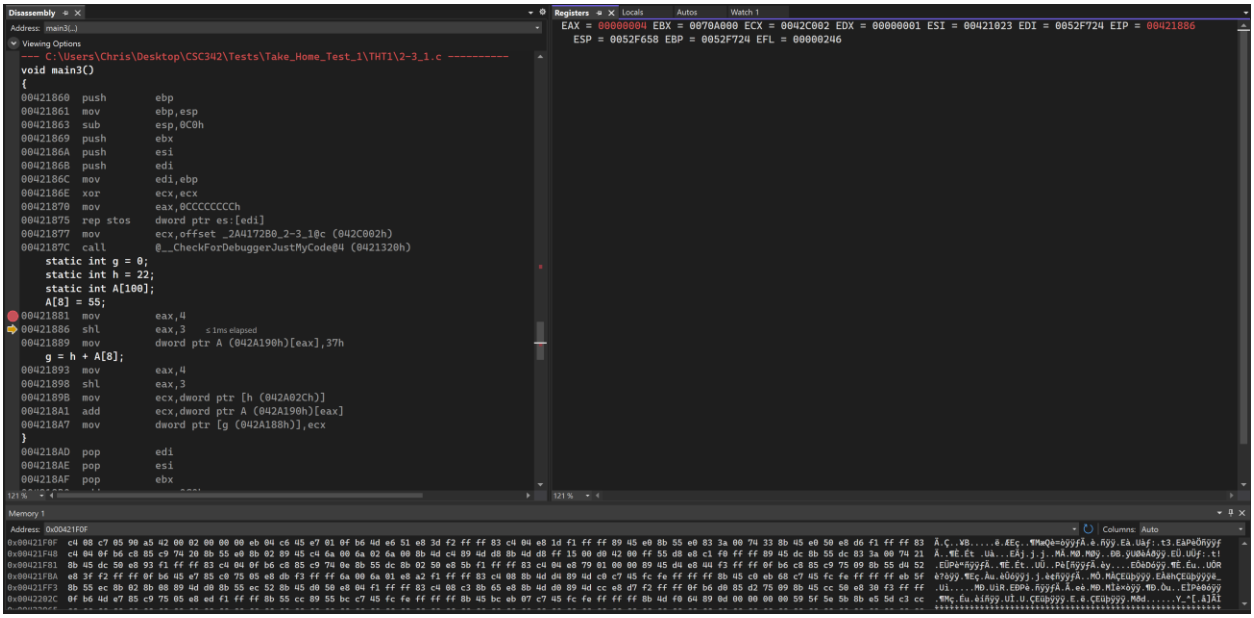


Figure 44. EAX with 4 moved into it

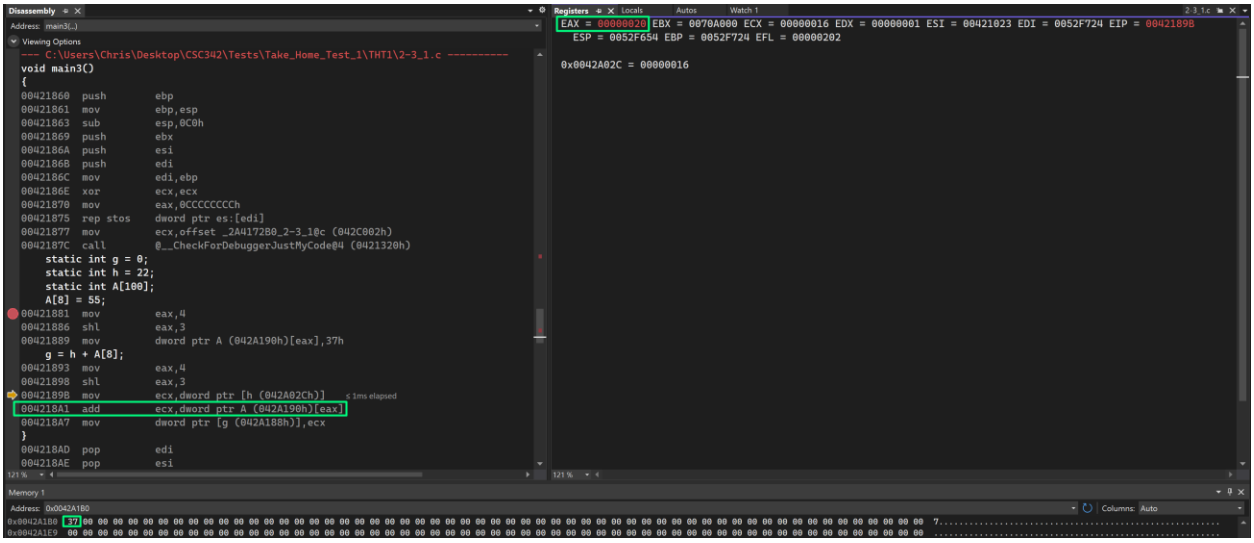


Figure 45. Shift to left,store 55 in 8th index

In the figure above, we can see that shifting the bits in the register to the left does the same task and multiplying 2 to 3rd index which is also 8. It causes the register EAX to have 32 which is equivalent to 20 in hex. We then store this 37H which is 55, into memory for the array. However, we store it in the array with offset of 32 from base address of A.

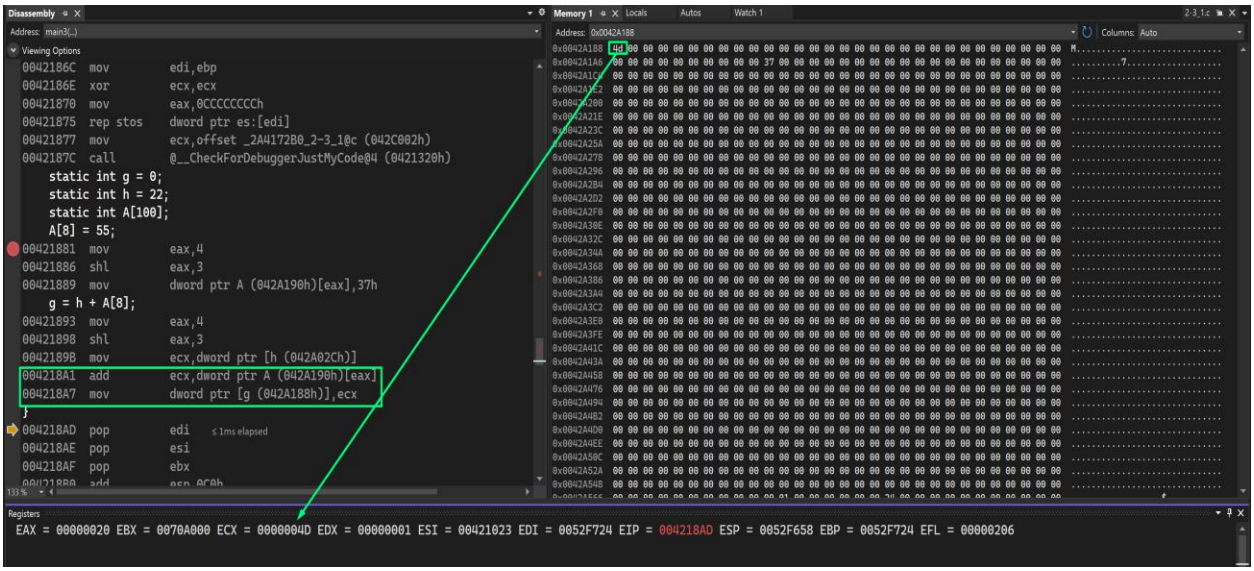


Figure 46. move + add op on arrary and h

The value of the 8th array element, 55, is added to h, which is saved in register ECX, and then relocated to the variable g, which is memory address 0x05A140h.

2-3_2.c

```
void main4()
{
    static int h = 25;
    static int A[100];
    A[8] = 200;
    A[12] = h + A[8];
}
```

Figure 47. 2-3_2.c code

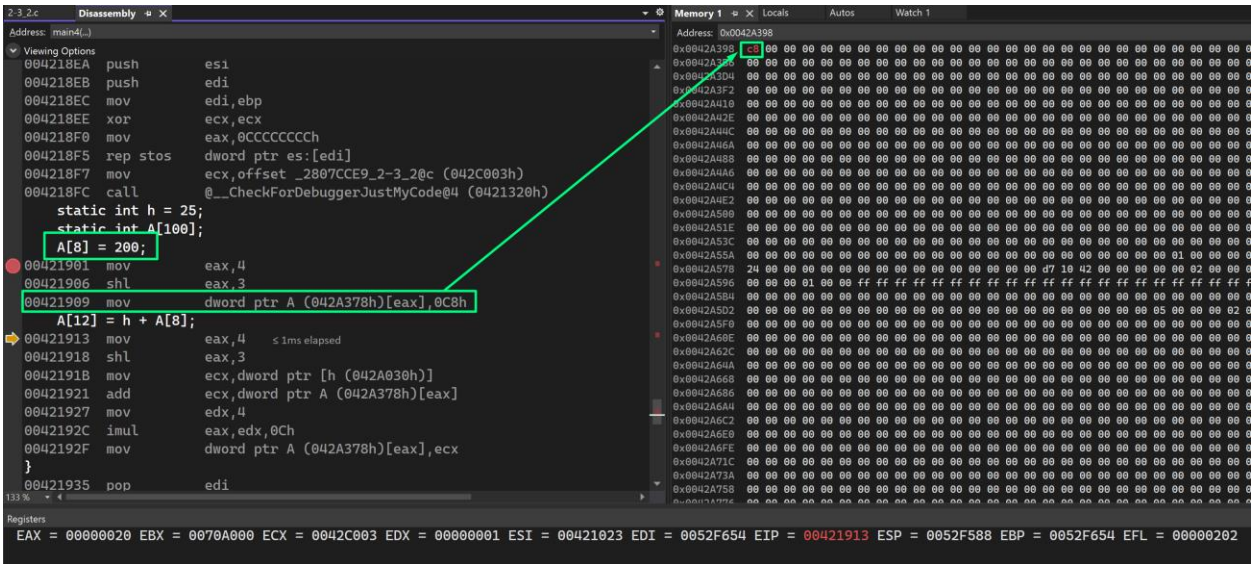


Figure 48. 200 stored in array which is in memory.

In this figure, 200 is set into memory address.

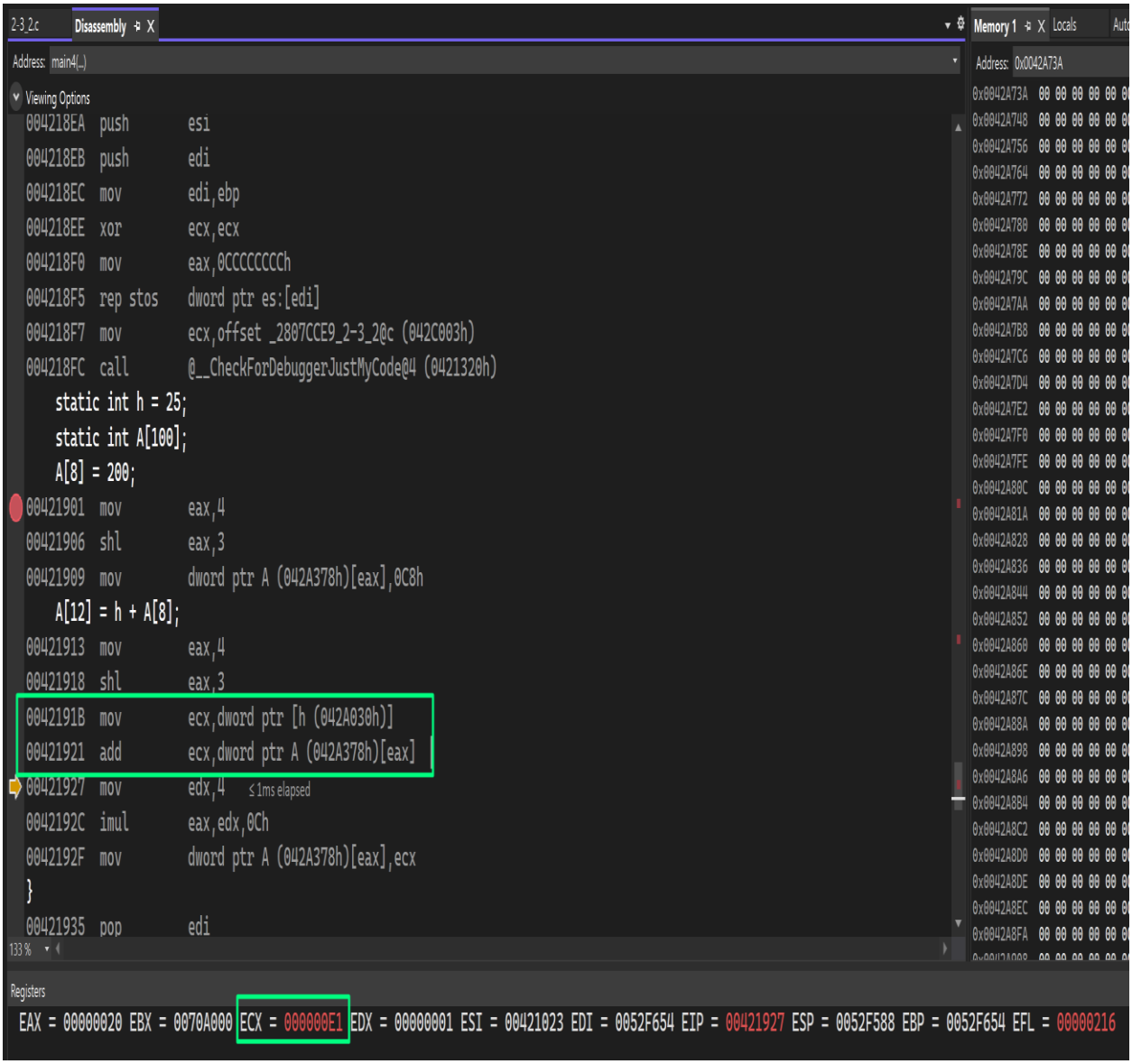


Figure 49. h + A[8] element stored into ECX

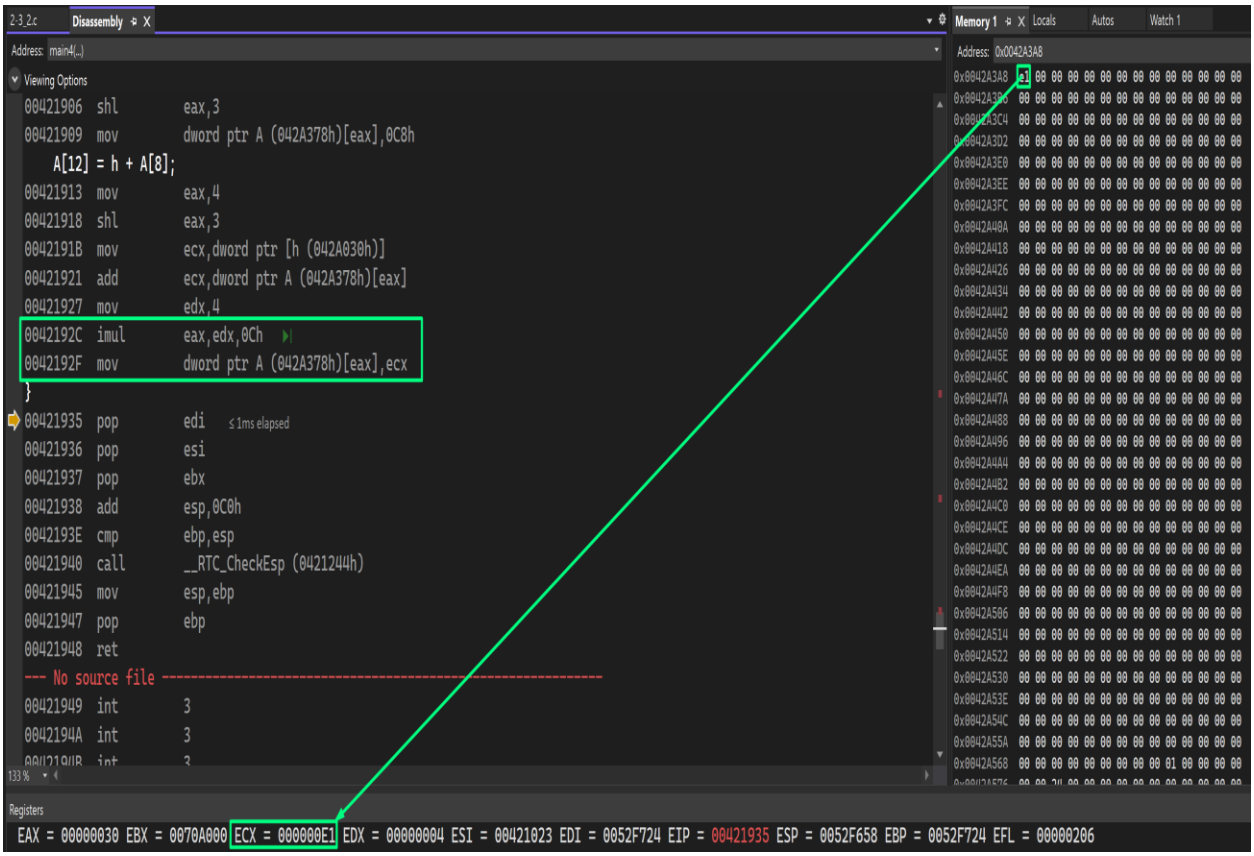


Figure 50. stored in array in 12th index

Here, the value was stored as seen in memory window.

2-5_1.c

```
void main5()
{
    static int a = 0;
    static int b = 0;
    static int c = 0;
    a = b + c;
}
```

Figure 51. 2-5_1.c code

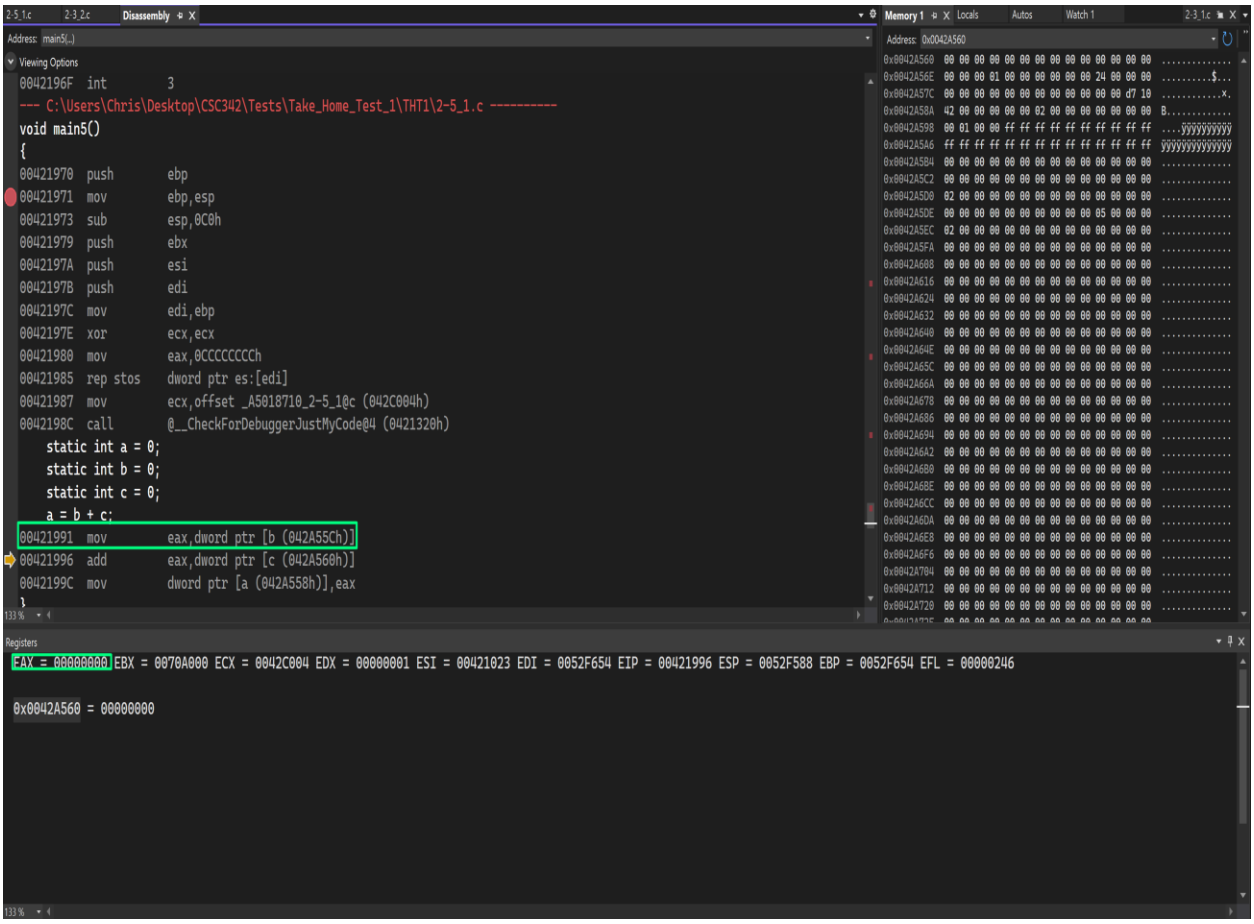


Figure 52. value of EAX is 0.

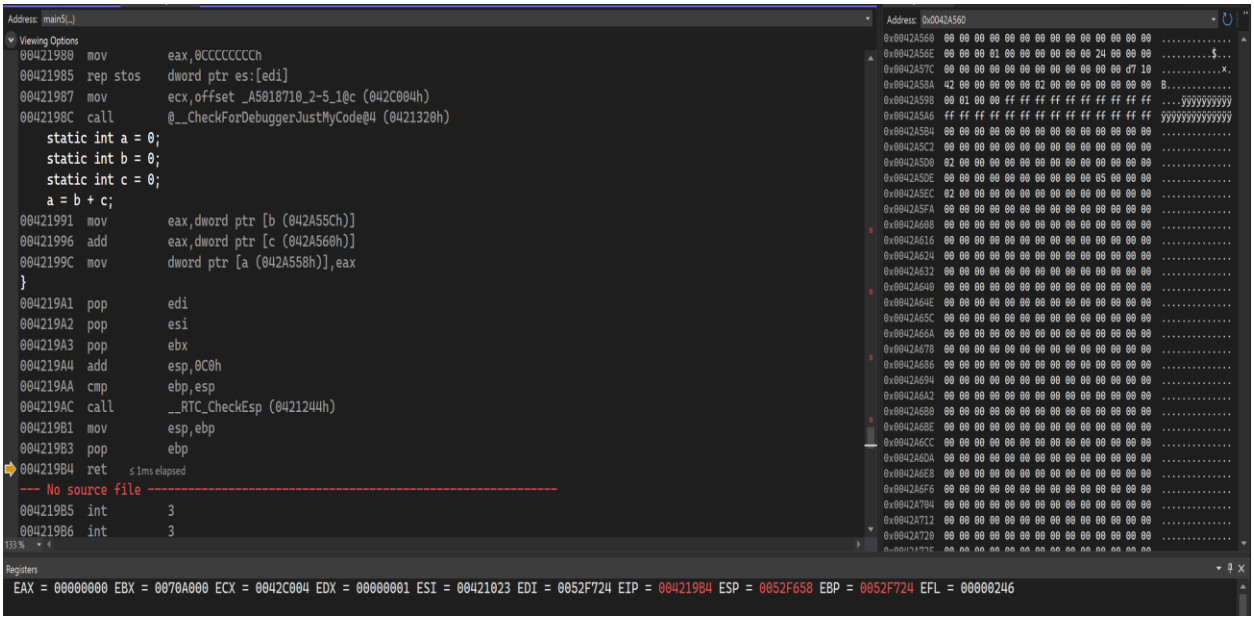


Figure 53. add + mov instruction on a.

We see that variable c is added into the contents of register EAX and then the result is then moved into a from EAX in memory.

2-6_1.c

```
void main6()
{
    static int s0 = 9;
    static int t1 = 0x3c00;
    static int t2 = 0xdc0;
    static int t3 = 0;

    t3 = s0 << 4;
    static int t0 = 0;
    t0 = t1 & t2;
    t0 = t1 | t2;
    t0 = ~t1;
}
```

Figure 54. 2-6_1.c code

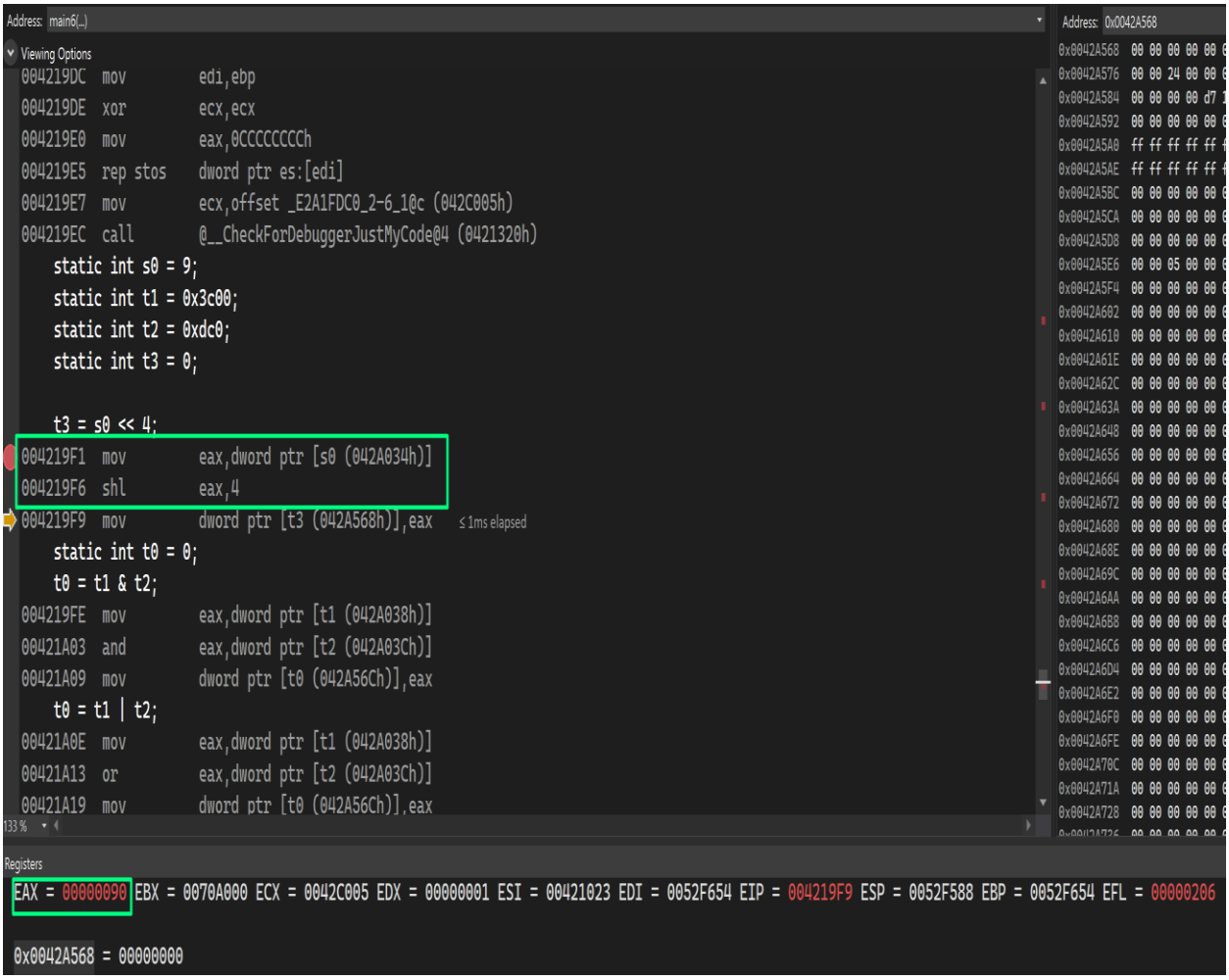


Figure 55. 9 stored in EAX, shifted left 4 bits.

Here, instructions allow the program to move 9 into EAX. From there eax, 4 tells us that it is shifted 4 bits to the left resulting in 0x90.

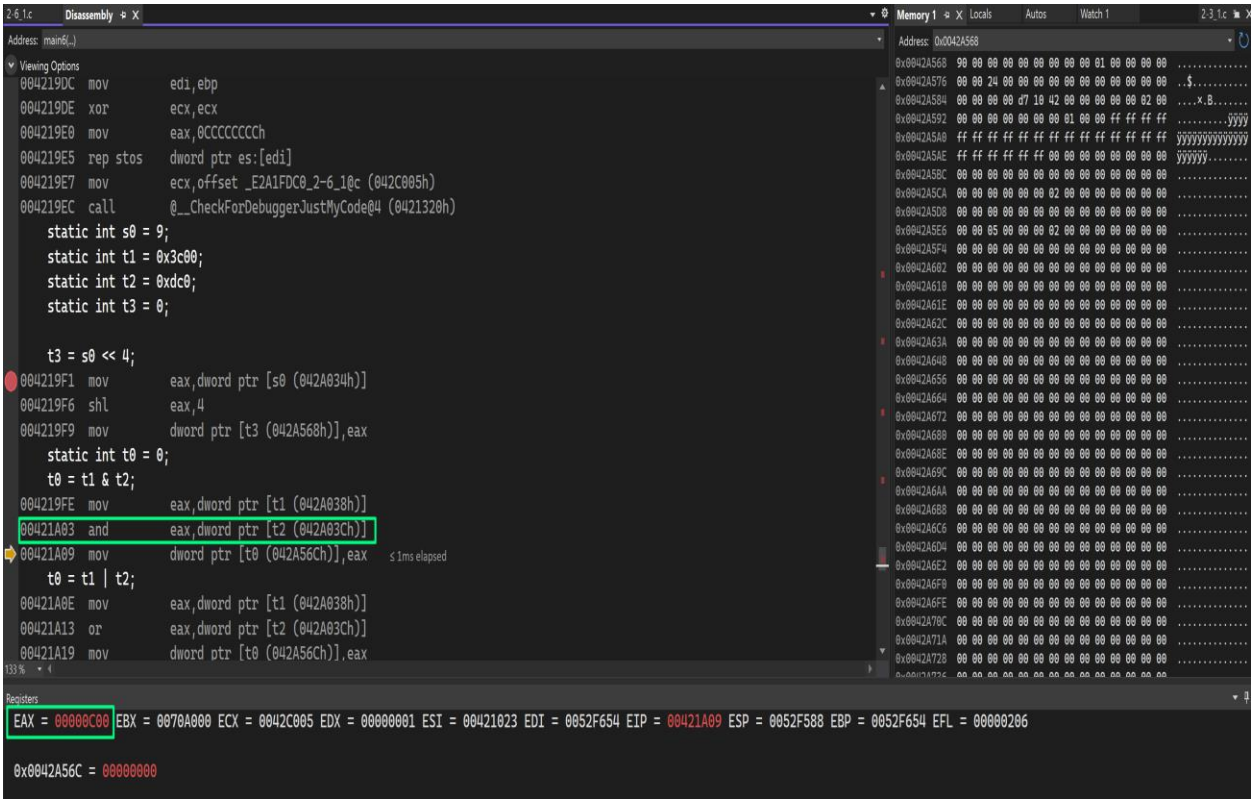


Figure 56. eax with and performed on it.

Variables t1 and t2 has had logical and operand performed on the EAX register in the figure above. See screenshot.

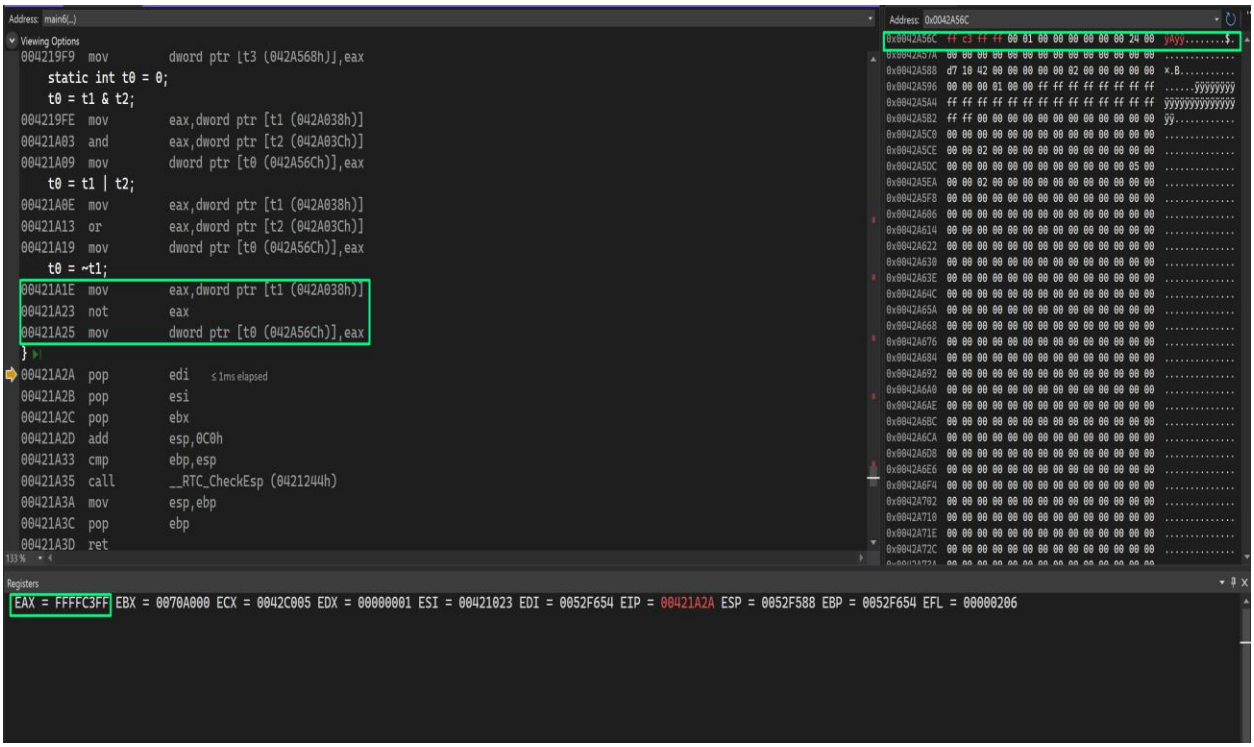


Figure 57. not operand on EAX reg.

As shown in the diagram above, variable t1 is transferred into register EAX, followed by a not operation on register EAX, and the contents of the EAX register are then put into variable t0 and stored in memory.

Natural_generator.c

```c
#include <stdio.h>
int natural_generator()
{
    int a = 1;
    static int b = -1;
    b += 1;
    return a + b;
}
int main1_1()
{
    printf("%d\n", natural_generator());
    printf("%d\n", natural_generator());
    printf("%d\n", natural_generator());
    return 0;
}
```

Figure 58. natural_generator.c code

```
void main()
{
00FF2070  push        ebp
00FF2071  mov         ebp,esp
00FF2073  sub         esp,0C0h
00FF2079  push        ebx
00FF207A  push        esi
00FF207B  push        edi
00FF207C  mov         edi,ebp
00FF207E  xor         ecx,ecx
00FF2080  mov         eax,0CCCCCCCCh
00FF2085  rep stos    dword ptr es:[edi]
00FF2087  mov         ecx,offset _17215B00_2-2_1@c (0FFC00Eh)
00FF208C  call        @__CheckForDebuggerJustMyCode@4 (0FF1320h)
    printf("%d\n", natural_generator());
00FF2091  call        _natural_generator (0FF13C5h)
00FF2096  push        eax
00FF2097  push        offset string "%d\n" (0FF7BCCh)
00FF209C  call        _printf (0FF13B6h)
00FF20A1  add         esp,8
    printf("%d\n", natural_generator());
00FF20A4  call        _natural_generator (0FF13C5h)
00FF20A9  push        eax
00FF20AA  push        offset string "%d\n" (0FF7BCCh)
00FF20AF  call        _printf (0FF13B6h)
00FF20B4  add         esp,8
    printf("%d\n", natural_generator());
00FF20B7  call        _natural_generator (0FF13C5h)
00FF20BC  push        eax
00FF20BD  push        offset string "%d\n" (0FF7BCCh)
00FF20C2  call        _printf (0FF13B6h)
00FF20C7  add         esp,8
```

Figure 59. natural_generator.c disassembly pt 1

```
}
00FF20CA  xor         eax,eax
00FF20CC  pop         edi
00FF20CD  pop         esi
00FF20CE  pop         ebx
00FF20CF  add         esp,0C0h
00FF20D5  cmp         ebp,esp
00FF20D7  call        __RTC_CheckEsp (0FF1244h)
00FF20DC  mov         esp,ebp
00FF20DE  pop         ebp
00FF20DF  ret
```

Figure 60. natural_generator.c disassembly pt 2

```
EAX = 011F5718 EBX = 00B47000 ECX = 00000001 EDX = 011F7D70 ESI = 00FF1023 EDI = 00FF1023 EIP = 00FF2070 ESP = 00D6FBF0 EBP = 00D6FC0C EFL = 00000202
```

Figure 61. register for natural_generator.c

While.c

```
void main13() {
    static int i = 1;
    static int product = 0;
    while (i < 5) {
        product = product * i;
        ++i;
    }
}
```

Figure 62. while.c code

```
void main() {
009B1780  push        ebp
009B1781  mov         ebp,esp
009B1783  sub         esp,0C0h
009B1789  push        ebx
009B178A  push        esi
009B178B  push        edi
009B178C  mov         edi,ebp
009B178E  xor         ecx,ecx
009B1790  mov         eax,0CCCCCCCCh
009B1795  rep stos    dword ptr es:[edi]
009B1797  mov         ecx,offset _17215B00_2-2_1@c (09BC006h)
009B179C  call        @__CheckForDebuggerJustMyCode@4 (09B1320h)
    static int i = 1;
    static int product = 0;
    while (i < 5) {
009B17A1  cmp         dword ptr [i (09BA000h)],5
009B17A8  jge         __$EncStackInitStart+3Eh (09B17CAh)
        product = product * i;
009B17AA  mov         eax,dword ptr [product (09BA184h)]
009B17AF  imul        eax,dword ptr [i (09BA000h)]
009B17B6  mov         dword ptr [product (09BA184h)],eax
        ++i;
009B17BB  mov         eax,dword ptr [i (09BA000h)]
009B17C0  add         eax,1
009B17C3  mov         dword ptr [i (09BA000h)],eax
    }
009B17C8  jmp         __$EncStackInitStart+15h (09B17A1h)
}
```

Figure 63. while.c disassembly pt 1

```
}
009B17CA  xor         eax,eax
009B17CC  pop         edi
009B17CD  pop         esi
009B17CE  pop         ebx
009B17CF  add         esp,0C0h
009B17D5  cmp         ebp,esp
009B17D7  call        __RTC_CheckEsp (09B1244h)
009B17DC  mov         esp,ebp
009B17DE  pop         ebp    ≤ 1ms elapsed
009B17DF  ret
```

Figure 64. while.c disassembly pt 2

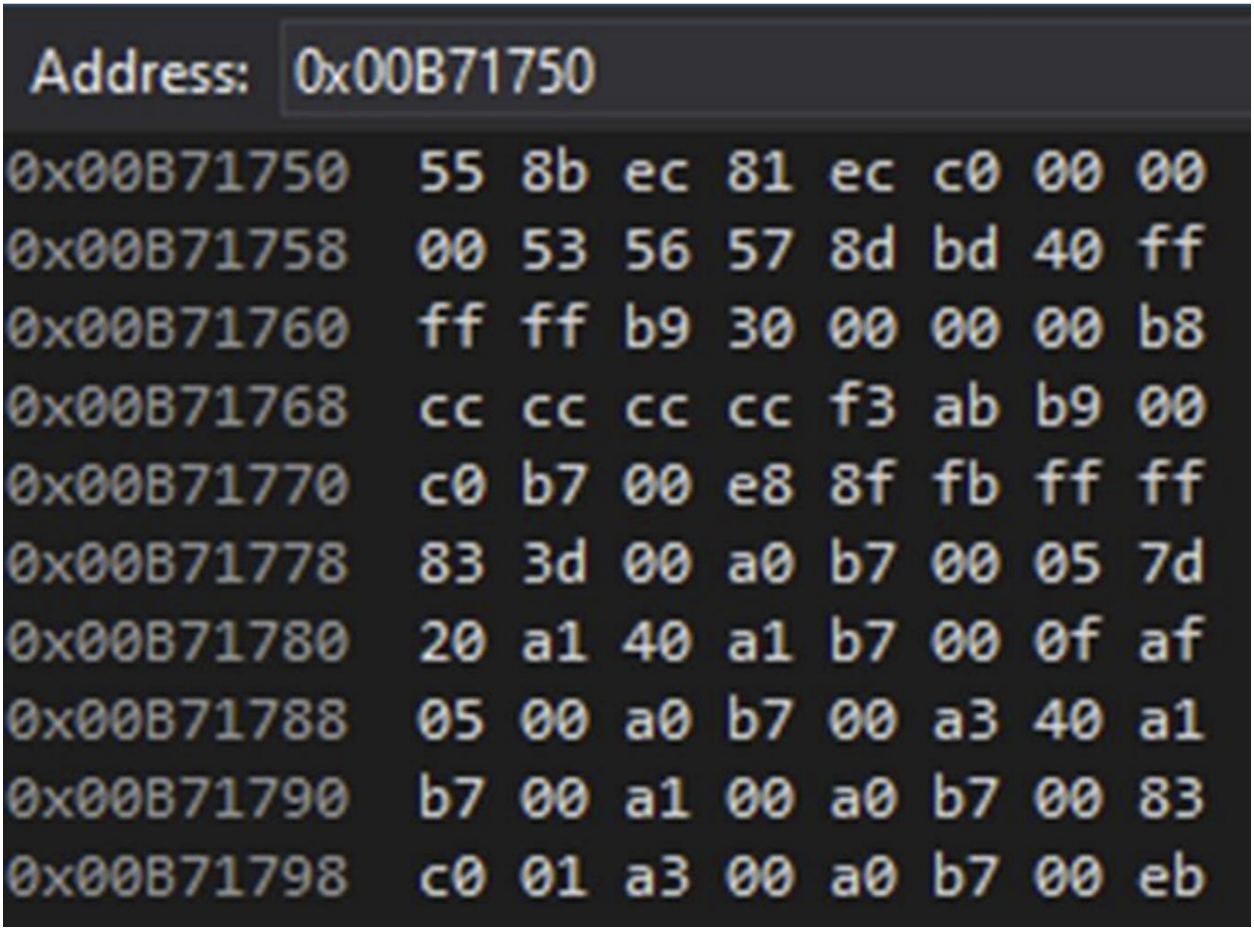


Figure 65. while.c memory

Figure 66. Example of registers.

For.c

```
void main10() {
    static int i = 1;
    static int product = 0;

    for (int i = 0; i < 5; ++i) {
        product = product * i;
    }
}
```

Figure 67. for.c code

```
void main() {
00E31780  push        ebp
00E31781  mov         ebp,esp
00E31783  sub         esp,0CCh
00E31789  push        ebx
00E3178A  push        esi
00E3178B  push        edi
00E3178C  lea         edi,[ebp-0Ch]
00E3178F  mov         ecx,3
00E31794  mov         eax,0CCCCCCCCh
00E31799  rep stos    dword ptr es:[edi]
00E3179B  mov         ecx,offset _17215B00_2-2_1@c (0E3C006h)
00E317A0  call        @__CheckForDebuggerJustMyCode@4 (0E31320h)
    static int i = 1;
    static int product = 0;

    for (int i = 0; i < 5; ++i) {
00E317A5  mov         dword ptr [ebp-8],0
00E317AC  jmp         __$EncStackInitStart+2Bh (0E317B7h)
00E317AE  mov         eax,dword ptr [ebp-8]
00E317B1  add         eax,1    ≤ 1ms elapsed
00E317B4  mov         dword ptr [ebp-8],eax
00E317B7  cmp         dword ptr [ebp-8],5
00E317BB  jge         __$EncStackInitStart+41h (0E317CDh)
        product = product * i;
00E317BD  mov         eax,dword ptr [product (0E3A184h)]
00E317C2  imul        eax,dword ptr [ebp-8]
00E317C6  mov         dword ptr [product (0E3A184h)],eax
    }
```

Figure 68. for.c disassembly pt 1

```
00E317CD  xor         eax,eax
00E317CF  pop         edi
00E317D0  pop         esi
00E317D1  pop         ebx
00E317D2  add         esp,0CCh
00E317D8  cmp         ebp,esp
00E317DA  call        __RTC_CheckEsp (0E31244h)
00E317DF  mov         esp,ebp
00E317E1  pop         ebp
00E317E2  ret
```

Figure 69. for.c disassembly pt 2

```
Registers
EAX = 00000000 EBX = 00AB2000 ECX = 00E3C006 EDX = 00000001 ESI = 00E31023 EDI = 00CFFBA0 EIP = 00E317B1 ESP = 00CFFAC8 EBP = 00CFFBA0 EFL = 00000206
```

Figure 70. register example for this code

Myadd.c

```
int My_Add(int num1, int num2) {
    return num1 + num2;
}

void main12() {
    int x = 2;
    int y = 3;
    My_Add(x, y);
}
```

Figure 71. myadd.c code

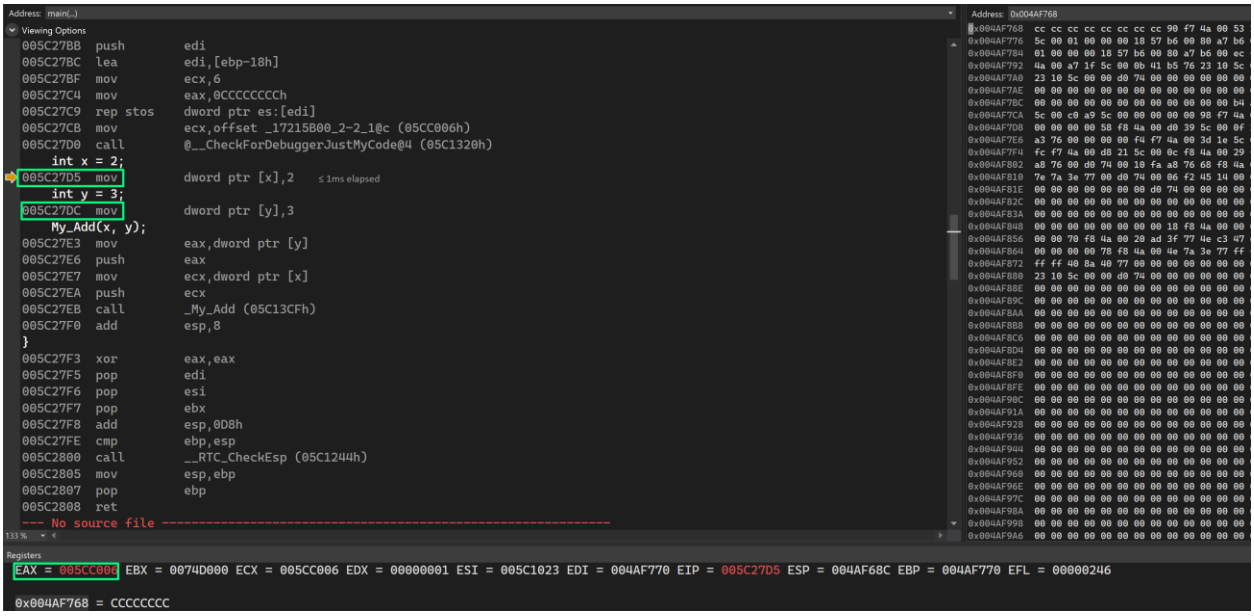


Figure 72. disassembly where mov is moving the integers to location.

Part 3: Intel x86_64 ISA, Linux 64bit (GDB)

The images below for Part 3 show the identical C code from Part 2 in GDB, as well as its disassembly, memory, and registers. I'll go over the components of a disassembly, memory, and registers windows using 2-2 1.c as an example, but I'll go over the parts of a disassembly, memory, and registers windows separately.

2-2_1.c

```
(gdb) disassemble
Dump of assembler code for function main:
   0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
=> 0x0000555555555131 <+8>:     mov    0x2ed9(%rip),%edx        # 0x555555558010 <b.4>
   0x0000555555555137 <+14>:    mov    0x2ed7(%rip),%eax        # 0x555555558014 <c.3>
   0x000055555555513d <+20>:    add    %edx,%eax
   0x000055555555513f <+22>:    mov    %eax,0x2ed3(%rip)        # 0x555555558018 <a.2>
   0x0000555555555145 <+28>:    mov    0x2ecd(%rip),%eax        # 0x555555558018 <a.2>
   0x000055555555514b <+34>:    mov    0x2ecb(%rip),%edx        # 0x55555555801c <e.1>
   0x0000555555555151 <+40>:    sub    %edx,%eax
   0x0000555555555153 <+42>:    mov    %eax,0x2ec7(%rip)        # 0x555555558020 <d.0>
--Type <RET> for more, q to quit, c to continue without paging--print /x $rbp
   0x0000555555555159 <+48>:    nop
   0x000055555555515a <+49>:    pop    %rbp
   0x000055555555515b <+50>:    ret
End of assembler dump.
(gdb) print /x $rsp
$1 = 0x7fffffffdd90
(gdb) print /x $rbp
$2 = 0x7fffffffdd90
```

Figure 73. RSP and RBP pointers

The address of the base point is put into the stack, and the address of the $rsp is copied into $rbp.

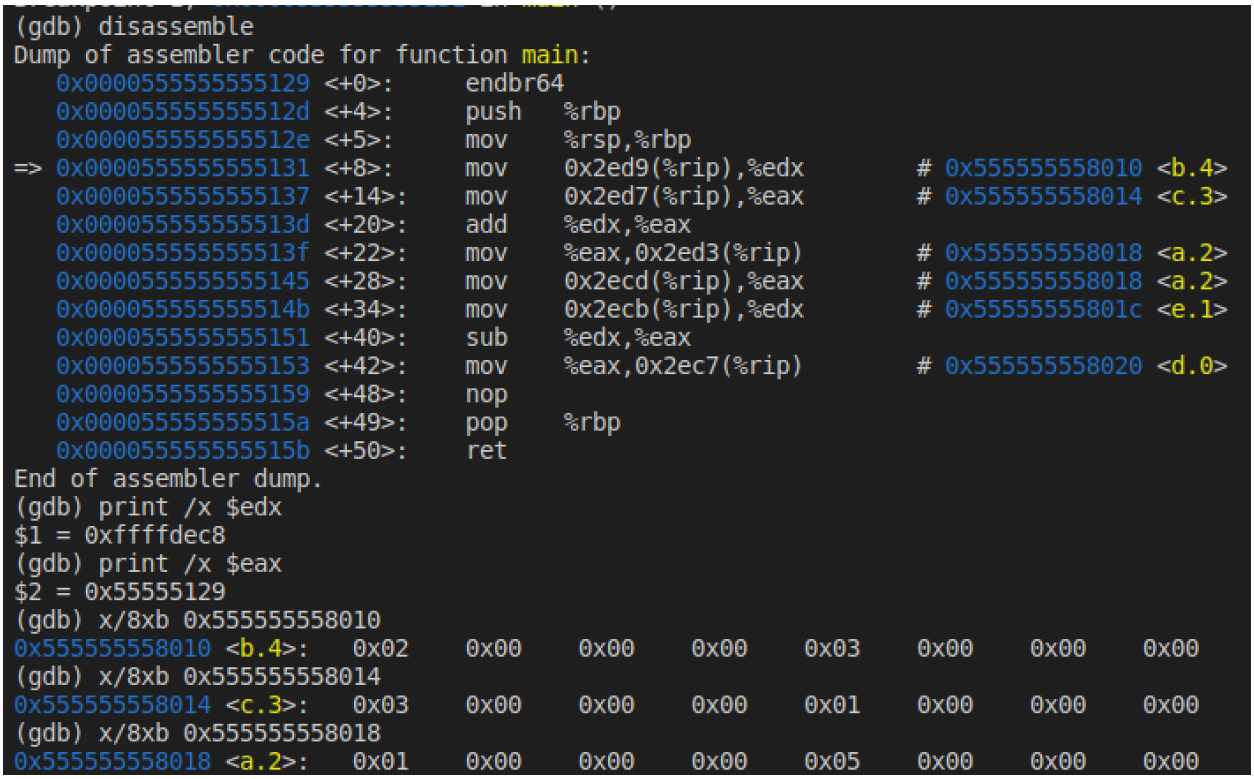


Figure 74. add operation on b+c and result stored in a.

From figure above, we see that register %edx has the value 2 for variable b. Variable be is stored at the address location of 0x555555558010 and variable c is located at address location 0x55555558014 and those values are stored in the registers %edx and %eax, respectively. The add instruction is then executed and the contents of registers %edx and %eax are added together and the result is stored in %eax. We then move the result into the address location where a is stored. The address location is 0x555555558018. From the figure we see where each variable is stored in memory.

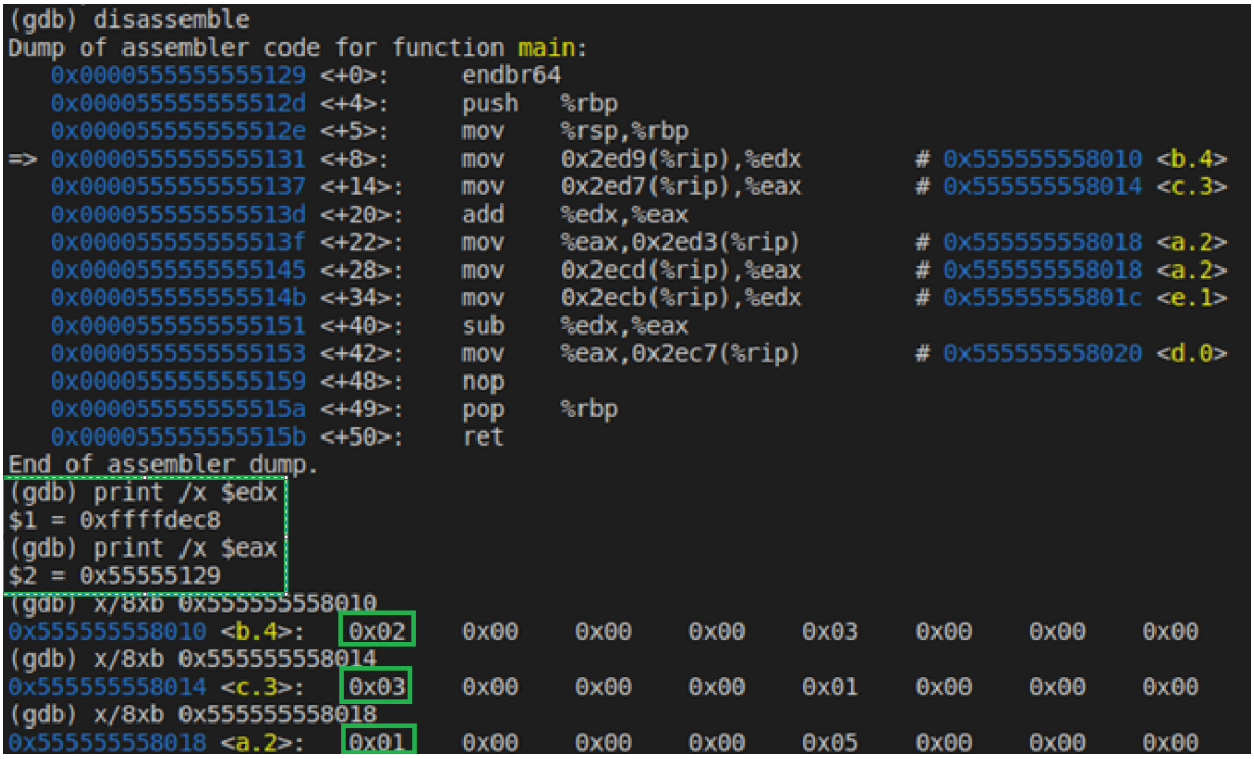


Figure 75. subtraction operation on a-e and stored in d

From figure above, we see that variable a and e both are obtained from their respective location in memory and added moved into registers %eax and %edx respectively. Then we perform the subtraction operation here but note that the subtraction operation doesn't occur until the arrow on the disassembly code is all the way at the end where the result is loved back into the memory where variable d is stored.

2-2_2.c

```
(gdb) disassemble
Dump of assembler code for function main:
   0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
=> 0x0000555555555131 <+8>:     mov    0x2ed9(%rip),%edx        # 0x555555558010 <g.4>
   0x0000555555555137 <+14>:    mov    0x2ed7(%rip),%eax        # 0x555555558014 <h.3>
   0x000055555555513d <+20>:    add    %edx,%eax
   0x000055555555513f <+22>:    mov    0x2ed3(%rip),%ecx        # 0x555555558018 <i.2>
   0x0000555555555145 <+28>:    mov    0x2ed1(%rip),%edx        # 0x55555555801c <j.1>
   0x000055555555514b <+34>:    add    %ecx,%edx
--Type <RET> for more, q to quit, c to continue without paging--print /x $edx
   0x000055555555514d <+36>:    sub    %edx,%eax
   0x000055555555514f <+38>:    mov    %eax,0x2ecf(%rip)        # 0x555555558024 <f.0>
   0x0000555555555155 <+44>:    nop
   0x0000555555555156 <+45>:    pop    %rbp
   0x0000555555555157 <+46>:    ret
End of assembler dump.
(gdb) print /x $edx
$1 = 0xffffdec8
(gdb) print /x $eax
$2 = 0x55555129
(gdb) print /x $ecx
$3 = 0x0
(gdb) x/8xb 0x555555558010
0x555555558010 <g.4>:   0x32    0x00    0x00    0x00    0x28    0x00    0x00    0x00
(gdb) x/8xb 0x555555558014
0x555555558014 <h.3>:   0x28    0x00    0x00    0x00    0x1e    0x00    0x00    0x00
(gdb) x/8xb 0x555555558018
0x555555558018 <i.2>:   0x1e    0x00    0x00    0x00    0x14    0x00    0x00    0x00
(gdb) x/8xb 0x55555555801c
0x55555555801c <j.1>:   0x14    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) x/8xb 0x555555558024
0x555555558024 <f.0>:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

Figure 76. disassemble code of 2-2_2.c

The values of each variable are taken from memory and saved in the registers, as shown in the diagram above. To begin, we can see that g+h is being added, with the values of g and h being grabbed from their respective memory locations and moved to the percent edx and percent eax registers. The result is kept in register percent eax once these registers are added together. After that, the i+j add operation is executed. The values of I and j are taken from memory, put in the appropriate registers, and then added together, as previously. Finally, the result of

the two registers, %edx and %eax are subtracted together to get the final answer that is stored in the %eax register and that value is then stored in the memory location of f, which is 0x555555524. We can see in the figure that the value has been stored in the location of variable f.

2-3_1

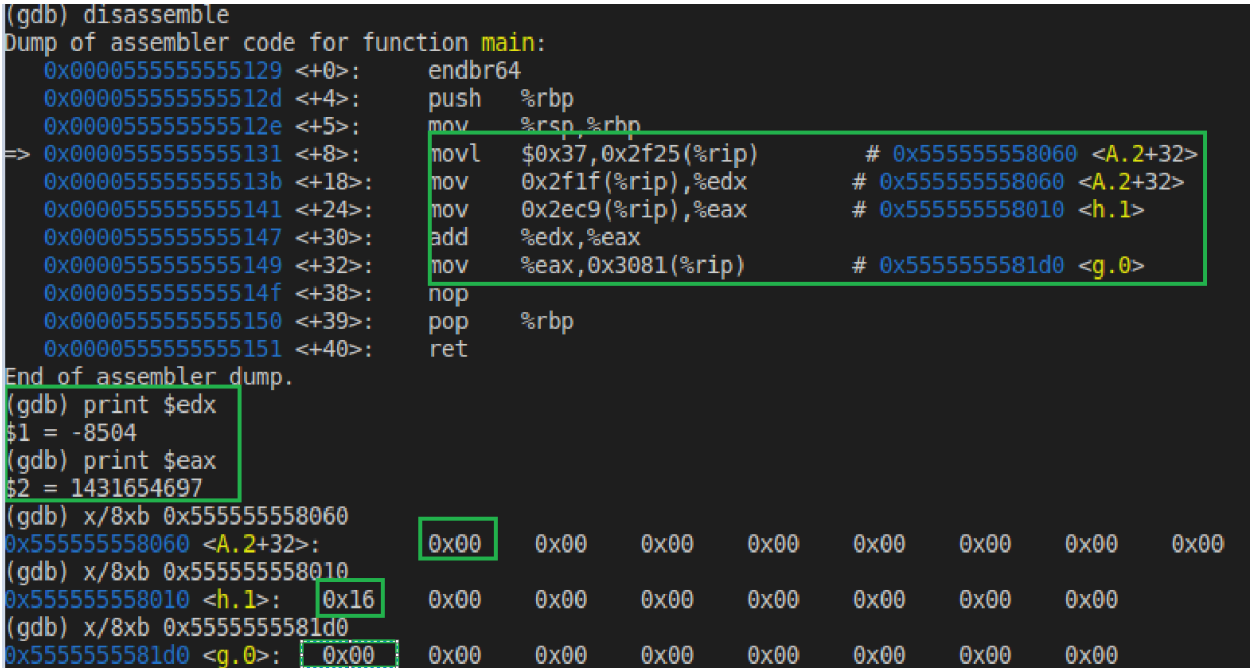


Figure 77. disassembly for 2-3_1.c

From figure above, we can see that we movl the value into the offset from the base address of array A. The value is stored in the array in the array at the address 0x555555558060. We can also see from the figure that value is loaded from memory into register %edx and the value h is loaded from address 0x555555558010 into the register %eax. These registers now have the value stored in h and they are added and the result is stored in register %eax. The result is then stored in memory where variable g is stored.

2-3_2.c

```
(gdb) disassemble
Dump of assembler code for function main:
   0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
=> 0x0000555555555131 <+8>:     movl   $0xc8,0x2f25(%rip)        # 0x555555558060 <A.1+32>
   0x000055555555513b <+18>:    mov    0x2f1f(%rip),%edx        # 0x555555558060 <A.1+32>
   0x0000555555555141 <+24>:    mov    0x2ec9(%rip),%eax        # 0x555555558010 <h.0>
   0x0000555555555147 <+30>:    add    %edx,%eax
   0x0000555555555149 <+32>:    mov    %eax,0x2f21(%rip)        # 0x555555558070 <A.1+48>
   0x000055555555514f <+38>:    nop
   0x0000555555555150 <+39>:    pop    %rbp
   0x0000555555555151 <+40>:    ret
End of assembler dump.
(gdb) print /x $edx
$1 = 0xffffdec8
(gdb) print /x $eax
$2 = 0x55555129
(gdb) x/8xb 0x555555558060
0x555555558060 <A.1+32>:        0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) x/8xb 0x555555558010
0x555555558010 <h.0>:   0x19    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) x/8xb 0x555555558070
0x555555558070 <A.1+48>:        0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

Figure 78. disassemble of 2-3_2.c

As seen in the diagram above, we store the value in memory with an offset from the array A's base address. This also signifies that we're adding as the array's eighth element. We then load that value into register percent edx from the array, as well as the value of h from address 0x555555558010 from register percent eax. The result of adding the two registers is kept in register percent eax. After that, the value in register percent eax is put in array A with an offset from the array's base address. This also means that we are adding the result to the 12 elements of the array A at the address of 0x55555555070. We see all these values stored in memory.

2-5_1.c

```
(gdb) disassemble
Dump of assembler code for function main:
   0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
=> 0x0000555555555131 <+8>:     mov    0x2edd(%rip),%edx        # 0x555555558014 <b.2>
   0x0000555555555137 <+14>:    mov    0x2edb(%rip),%eax        # 0x555555558018 <c.1>
   0x000055555555513d <+20>:    add    %edx,%eax
   0x000055555555513f <+22>:    mov    %eax,0x2ed7(%rip)        # 0x55555555801c <a.0>
   0x0000555555555145 <+28>:    nop
   0x0000555555555146 <+29>:    pop    %rbp
   0x0000555555555147 <+30>:    ret
End of assembler dump.
(gdb) print $edx
$1 = -8504
(gdb) print $eax
$2 = 1431654697
(gdb) print /x $edx
$3 = 0xffffdec8
(gdb) print /x $eax
$4 = 0x55555129
(gdb) x/8xb 0x555555558014
0x555555558014 <b.2>:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) x/8xb 0x555555558018
0x555555558018 <c.1>:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) x/8xb 0x55555555801c
0x55555555801c <a.0>:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

Figure 79. disassemble of 2-5_1.c

2-6_1.c

```
(gdb) disassemble
Dump of assembler code for function main:
   0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
=> 0x0000555555555131 <+8>:     mov    0x2ed9(%rip),%eax        # 0x555555558010 <s0.4>
   0x0000555555555137 <+14>:    shl    $0x4,%eax
   0x000055555555513a <+17>:    mov    %eax,0x2ee0(%rip)        # 0x555555558020 <t3.3>
   0x0000555555555140 <+23>:    mov    0x2ece(%rip),%edx        # 0x555555558014 <t1.2>
   0x0000555555555146 <+29>:    mov    0x2ecc(%rip),%eax        # 0x555555558018 <t2.1>
   0x000055555555514c <+35>:    and    %edx,%eax
   0x000055555555514e <+37>:    mov    %eax,0x2ed0(%rip)        # 0x555555558024 <t0.0>
   0x0000555555555154 <+43>:    mov    0x2eba(%rip),%edx        # 0x555555558014 <t1.2>
   0x000055555555515a <+49>:    mov    0x2eb8(%rip),%eax        # 0x555555558018 <t2.1>
   0x0000555555555160 <+55>:    or     %edx,%eax
   0x0000555555555162 <+57>:    mov    %eax,0x2ebc(%rip)        # 0x555555558024 <t0.0>
   0x0000555555555168 <+63>:    mov    0x2ea6(%rip),%eax        # 0x555555558014 <t1.2>
   0x000055555555516e <+69>:    not    %eax
   0x0000555555555170 <+71>:    mov    %eax,0x2eae(%rip)        # 0x555555558024 <t0.0>
   0x0000555555555176 <+77>:    nop
   0x0000555555555177 <+78>:    pop    %rbp
   0x0000555555555178 <+79>:    ret
End of assembler dump.
(gdb) print /x $eax
$1 = 0x55555129
(gdb) print /x $edx
$2 = 0xffffdec8
```

Figure 80. disassemble of 2-6_1.c pt 1

```
(gdb) x/8xb 0x555555558014
0x555555558014 <t1.2>:  0x00    0x3c    0x00    0x00    0xc0    0x0d    0x00    0x00
(gdb) x/8xb 0x555555558010
0x555555558010 <s0.4>:  0x09    0x00    0x00    0x00    0x00    0x3c    0x00    0x00
(gdb) x/8xb 0x555555558020
0x555555558020 <t3.3>:  0x90    0x00    0x00    0x00    0xff    0xc3    0xff    0xff
(gdb) x/8xb 0x555555558018
0x555555558018 <t2.1>:  0xc0    0x0d    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) x/8xb 0x555555558024
0x555555558024 <t0.0>:  0xff    0xc3    0xff    0xff    0x00    0x00    0x00    0x00
```

Figure 81. disassemble of 2-6_1.c pt 2

Natural_generator.c

```
(gdb) disassemble
Dump of assembler code for function main:
   0x0000555555555174 <+0>:     endbr64
   0x0000555555555178 <+4>:     push   %rbp
   0x0000555555555179 <+5>:     mov    %rsp,%rbp
=> 0x000055555555517c <+8>:     mov    $0x0,%eax
   0x0000555555555181 <+13>:    call   0x555555555149 <natural_generator>
   0x0000555555555186 <+18>:    mov    %eax,%esi
   0x0000555555555188 <+20>:    lea    0xe75(%rip),%rax        # 0x555555556004
   0x000055555555518f <+27>:    mov    %rax,%rdi
   0x0000555555555192 <+30>:    mov    $0x0,%eax
   0x0000555555555197 <+35>:    call   0x555555555050 <printf@plt>
   0x000055555555519c <+40>:    mov    $0x0,%eax
   0x00005555555551a1 <+45>:    call   0x555555555149 <natural_generator>
   0x00005555555551a6 <+50>:    mov    %eax,%esi
--Type <RET> for more, q to quit, c to continue without paging--c
   0x00005555555551a8 <+52>:    lea    0xe55(%rip),%rax        # 0x555555556004
   0x00005555555551af <+59>:    mov    %rax,%rdi
   0x00005555555551b2 <+62>:    mov    $0x0,%eax
   0x00005555555551b7 <+67>:    call   0x555555555050 <printf@plt>
   0x00005555555551bc <+72>:    mov    $0x0,%eax
   0x00005555555551c1 <+77>:    call   0x555555555149 <natural_generator>
   0x00005555555551c6 <+82>:    mov    %eax,%esi
   0x00005555555551c8 <+84>:    lea    0xe35(%rip),%rax        # 0x555555556004
   0x00005555555551cf <+91>:    mov    %rax,%rdi
   0x00005555555551d2 <+94>:    mov    $0x0,%eax
   0x00005555555551d7 <+99>:    call   0x555555555050 <printf@plt>
   0x00005555555551dc <+104>:   mov    $0x0,%eax
   0x00005555555551e1 <+109>:   pop    %rbp
   0x00005555555551e2 <+110>:   ret
End of assembler dump.
(gdb) print /x $eax
$1 = 0x55555174
(gdb) print /x $edx
$2 = 0xffffdec8
```

Figure 82. disassemble of natural_generator.c

You can see in figure above that print is visible and call instruction being used.

This instruction calling the function natural generator and then storing the address of the new generated value.

Myadd.c

```
(gdb) disassemble
Dump of assembler code for function main:
   0x0000555555555141 <+0>:     endbr64
   0x0000555555555145 <+4>:     push   %rbp
   0x0000555555555146 <+5>:     mov    %rsp,%rbp
=> 0x0000555555555149 <+8>:     sub    $0x10,%rsp
   0x000055555555514d <+12>:    movl   $0x2,-0x8(%rbp)
   0x0000555555555154 <+19>:    movl   $0x3,-0x4(%rbp)
   0x000055555555515b <+26>:    mov    -0x4(%rbp),%edx
   0x000055555555515e <+29>:    mov    -0x8(%rbp),%eax
   0x0000555555555161 <+32>:    mov    %edx,%esi
   0x0000555555555163 <+34>:    mov    %eax,%edi
   0x0000555555555165 <+36>:    call   0x555555555129 <My_Add>
   0x000055555555516a <+41>:    nop
   0x000055555555516b <+42>:    leave
   0x000055555555516c <+43>:    ret
End of assembler dump.
(gdb) print /x $eax
$1 = 0x55555141
(gdb) print /x #edx
Invalid character '#' in expression.
(gdb) x/8xb 0x555555555129
0x555555555129 <My_Add>:        0xf3    0x0f    0x1e    0xfa    0x55    0x48  0x89     0xe5
```

Figure 83. myadd.c disassemble

Jle: conditional jump. It will come to this point check the condition and the bit size that is saved then move to the label that was assigned If the condition is true.

Lea: allows us to obtain the address after certain instructions are done.

Jmp: unconditional jump – it will come to this point and change the pointer registers and instructions to execute will change no matter what. This changes flow of instruction to another point.

Cmp: This compares two operands by basically subtracting it. If the subtraction is positive it returns a 0 and if it is negative it returns a 1. This is different from different assembly languages.

For.c

```
void main() {
    static int i = 1;
    static int product = 0;

    for (int i = 0; i < 5; ++i) {
        product = product * i;
    }
}
```

Figure 84. for.c code

```
(gdb) disassemble
Dump of assembler code for function main:
   0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
=> 0x0000555555555131 <+8>:     movl   $0x0,-0x4(%rbp)
   0x0000555555555138 <+15>:    jmp    0x55555555514e <main+37>
   0x000055555555513a <+17>:    mov    0x2ed8(%rip),%eax        # 0x555555558018 <product.1>
   0x0000555555555140 <+23>:    imul   -0x4(%rbp),%eax
   0x0000555555555144 <+27>:    mov    %eax,0x2ece(%rip)        # 0x555555558018 <product.1>
   0x000055555555514a <+33>:    addl   $0x1,-0x4(%rbp)
   0x000055555555514e <+37>:    cmpl   $0x4,-0x4(%rbp)
   0x0000555555555152 <+41>:    jle    0x55555555513a <main+17>
   0x0000555555555154 <+43>:    nop
   0x0000555555555155 <+44>:    nop
   0x0000555555555156 <+45>:    pop    %rbp
   0x0000555555555157 <+46>:    ret
End of assembler dump.
```

Figure 85. for.c disassemble

If.c

```
(gdb) disassemble
Dump of assembler code for function main:
   0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
=> 0x0000555555555131 <+8>:     mov    0x2edd(%rip),%eax        # 0x555555558014 <i.1>
   0x0000555555555137 <+14>:    cmp    $0x4,%eax
   0x000055555555513a <+17>:    jg     0x555555555148 <main+31>
   0x000055555555513c <+19>:    movl   $0xa,0x2ed2(%rip)        # 0x555555558018 <result.0>
   0x0000555555555146 <+29>:    jmp    0x555555555152 <main+41>
   0x0000555555555148 <+31>:    movl   $0xfffffff6,0x2ec6(%rip)        # 0x555555558018 <result.0>
   0x0000555555555152 <+41>:    nop
   0x0000555555555153 <+42>:    pop    %rbp
   0x0000555555555154 <+43>:    ret
End of assembler dump.
```

Figure 86. if.c disassemble

While.c

```
(gdb) disassemble
Dump of assembler code for function main:
   0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
=> 0x0000555555555131 <+8>:     jmp    0x555555555157 <main+46>
   0x0000555555555133 <+10>:    mov    0x2edf(%rip),%edx        # 0x555555558018 <product.1>
   0x0000555555555139 <+16>:    mov    0x2ed1(%rip),%eax        # 0x555555558010 <i.0>
   0x000055555555513f <+22>:    imul   %edx,%eax
   0x0000555555555142 <+25>:    mov    %eax,0x2ed0(%rip)        # 0x555555558018 <product.1>
   0x0000555555555148 <+31>:    mov    0x2ec2(%rip),%eax        # 0x555555558010 <i.0>
   0x000055555555514e <+37>:    add    $0x1,%eax
   0x0000555555555151 <+40>:    mov    %eax,0x2eb9(%rip)        # 0x555555558010 <i.0>
   0x0000555555555157 <+46>:    mov    0x2eb3(%rip),%eax        # 0x555555558010 <i.0>
   0x000055555555515d <+52>:    cmp    $0x4,%eax
   0x0000555555555160 <+55>:    jle    0x555555555133 <main+10>
   0x0000555555555162 <+57>:    nop
   0x0000555555555163 <+58>:    nop
   0x0000555555555164 <+59>:    pop    %rbp
   0x0000555555555165 <+60>:    ret
End of assembler dump.
```

Figure 87. while.c disassemble

Comparison:

Mips:

The MIPS MARS simulator is a 32-bit processor, and all of its instructions are 32 bits long. The length of the instructions remains constant throughout the duration of the program. Registers on the MIPS processor include zero, stack pointer, frame pointer, program counter, and so on. Because it points to the top of the stack in our register, the stack pointer is arguably the most critical to comprehend. The frame

pointer is pointing to the stack's bottom. Data is stored in Big Endian notation in the simulator. This essentially indicates that we can read data from left to right and that the highest byte is stored at the lowest memory address. The arithmetic the register to hold input, input one, and input two in the MARS MIPS simulator. You can store and, nor, or, and other logical bitwise operations in the logical register. The instructions "lw" and "sw," which stand for load word and store word, are used to communicate data between registers and memory. The load word copies data from memory to a register, whereas the store word copies data from the register back to memory. MIPS uses bne (branch not equal) instructions to build loop statements. The instruction decoder will jump to the designated label if the two operands are not equal. In MIPS, "jal" is used to invoke functions. This saves a return address for the registers to use when specific calls have been completed. The storage of variables in a MIPS processor is determined by the type of variables. Static variables are stored outside of the stack in memory and accessible via an offset from a base address, whereas local variables are stored on the stack and accessed via an offset from the stack pointer. Before the program begins to run, static variables are set. During runtime, local variables are initialized, and the software stores the coded values in a register.

Intel x32:

We used Visual Studios on Windows to analyze the disassembly for Intel using a 32-bit compiler and debugger. Intel's instruction sets are dynamic and change depending on which instruction is invoked. Small Endian notation is used to store data, which means the least significant byte is put in the lower memory address. Unlike MIPS, Intel simply requires one operand to be in a register, while the second operand can be stored in memory. The instruction "mov" is used to transfer data. Intel uses jne and jmp to create loops. In Intel, function calls are made with the call instruction, which sets the instruction pointer to an instruction that calls a function jump. When the function call is finished, the return address is saved into the stack to be used later. Outside of the stack, static variables are saved in memory. Local variables are assigned stack addresses, and values are loaded directly into memory rather than into a register.

Linux:

All platforms compare the values within the conditions by moving those values into registers and using comparison-based instructions. All platforms compare the values within the conditions by moving those values into registers and using comparison-based instructions. In and Linux, the while loop iterates by running through the code and having a jump instruction at the end of the iteration. Also whenever, the while loop iterates it has a conditional jump as the first line of the while-scope that is used to break out of the loop if certain conditions are not met.

All platforms compare the values within the conditions by moving those values into registers and using comparison-based instructions. In Linux, the for loop iterates by running through the code and having a jump instruction at the end of the iteration. Also, whenever the for-loop iterates, it has a conditional jump that is used to break out of the loop if certain conditions are not met.

Conclusion:

In this Take Home Test, I discovered that, despite the many architectures, operating systems, and processors, computers in general execute lines of code in a similar fashion. Furthermore, several instructions are performed during the execution of a program, and each architecture has its own style of ordering these statements. Registers and RAM are used by MIPS, Windows 32-bit Intel x86, and Linux Intel x86 with gcc/gdb. However, one significant difference between them is that Intel uses pointers instead of addresses in memory. In a similar vein, they all interact with stack pointers and registers in their own unique ways. In general, each of these architectures has taught me a great deal about how computers interpret instructions and how they are executed.