

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

- Anthony Ramos

Take Home Test #1

Comparison of Instruction Set Architecture (ISA)

Anthony Ramos

CSC342

Spring 2021

Professor Izidor Gertner

Start Time: March 31st 2021 @ 6:00pm

End Time: April 4rd @ 6:30pm

All source code used can be found [here](#).

Contents

Objective:	1
Part I: MIPS Assembly on MARS Simulator	2
2-2_1.asm (Simple Arithmetic)	3
2-3_1.asm (Arrays).....	16
2-5_2.asm (Arrays).....	23
2-6_1.asm (Bitwise Operations)	26
If_Else.asm.....	29
WhileLoop.asm.....	31
Local.asm	33
Natural_Generator.asm.....	35
Part II: Intel X86 on MS Visual Studio (32-Bit)	37
Debugging a C program in MS Visual Studio	38
2-2_1.c (Simple Arithmetic)	40
2-3_1.c (Arrays).....	43
2-5_2.c (Arrays).....	45
2-6_1.c (Bitwise Operations)	47
If_Else.c.....	51
WhileLoop.c.....	54
Local.c	57
Natural_Generator.c.....	60
Part III: Intel X86 64-bit on Linux platform using GDB	62
Debugging a C program in Linux using GDB	63
2-2_1.c (Simple Arithmetic)	64
2-3_1.c (Arrays).....	67
2-5_2.c (Arrays).....	69
2-6_1.c (Bitwise Operations)	70
If_Else.c.....	73
WhileLoop.c.....	75
Local.c	77
Natural_Generator.c.....	79

Objective: The objective of this test is to explore and demonstrate understanding of assembly code and machine instruction operations in three sets of architecture: MIPS instruction set architecture, Intel x86 ISA via MS Visual Studio 32-Bit compiler and debugger, and Intel X86 64-bit running Linux platform (64-bit GCC and GDB). In each part of this report, the same list of programs will be analyzed on each platform. Each program will aim to cover a C programming structure (i.e. loops, arrays, pointers, bitwise operations, etc.)

Part I: MIPS Assembly on MARS Simulator

**Topics Covered: Static/Local variables, Arrays, Bitwise Operations, Condition Statements,
Loops**

2-2_1.asm (Simple Arithmetic)

The following MIPS assembly program makes use of 5 *static* variables *a* through *e* to perform two arithmetic operations. The first is the addition of *b* and *c* and the second is the subtraction of *a* and *e*.

```
.data
a: .word 1
b: .word 2
c: .word 3
d: .word 4
e: .word 5

.text
lw $s0, a
lw $s1, b
lw $s2, c
lw $s3, d
lw $s4, e
# a = b + c
add $s0, $s1, $s2
sw $s0, a
# d = a - e
sub $s3, $s0, $s4
sw $s3, d
```

Figure 1: 2-2_1.asm

We execute this assembly program which produces the three windows in figure 2 below.

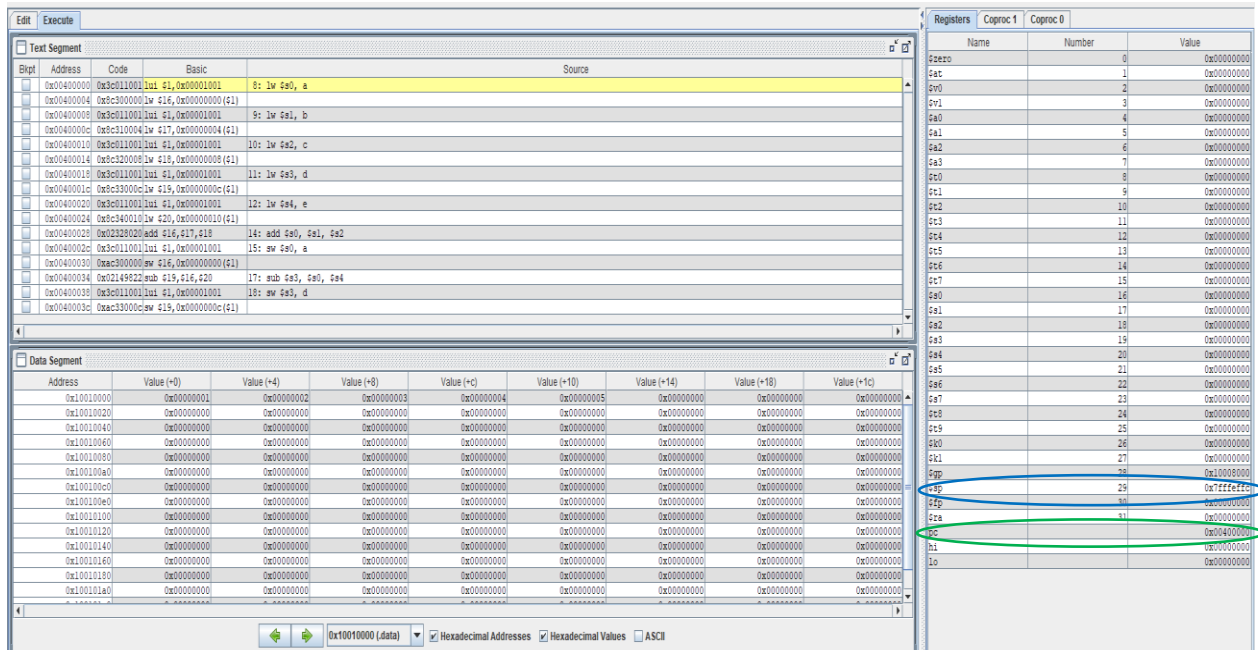
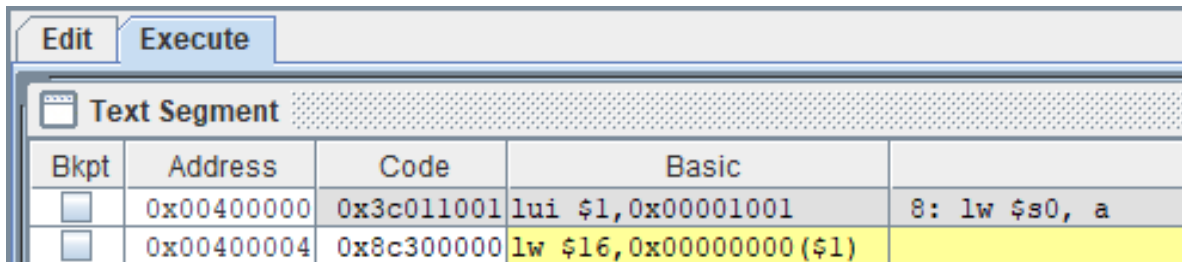


Figure 2: Initial Text Segment, Data Segment, and Register windows

Figure 2 shows the initial *Text Segment*, *Data Segment*, and *Register* windows. The first window is the Text Segment window which comprises of four columns: address, code, basic and source. These columns show the address of the instruction, the 32-Bit machine instruction, the MIPS instruction, and the assembly code respectively. The Register window displays all the registers along with their values in hexadecimal. The Data segment window contains the WORD stored in the respective memory addresses. It contains columns which displays the address value plus the offset indicated by (+*offset*).

Initially, they all have value zero except for the global pointer (\$gp), stack pointer (\$sp), and program counter (\$pc). The program counter initially has value 0x00400000 (circled in green), which points to the next instruction highlighted in yellow above. The stack pointer (circled in blue), which points to the top of the stack has an initial value of 0x7ffefffc. It is important to realize that, because all variables in this assembly code are all *static*, the value of the stack pointer will not change.

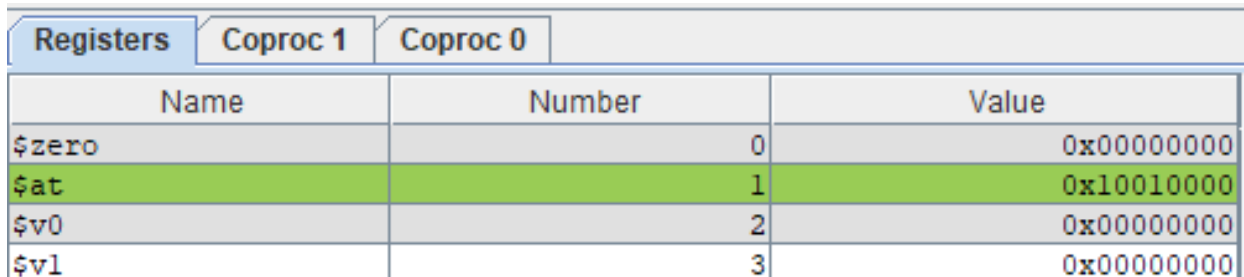
To run through the assembly program, we use the F7 key. Upon doing so, the Text Segment window is updated to figure 3a below.



Bkpt	Address	Code	Basic	
<input type="checkbox"/>	0x00400000	0x3c011001	lui \$1, 0x00001001	8: lw \$s0, a
<input type="checkbox"/>	0x00400004	0x8c300000	lw \$16, 0x00000000(\$1)	

Figure 3a: Text Segment window showing current instruction & next instruction (in yellow)

It shows that we are currently executing the instruction `lui $1, 0x00001001` whose purpose is to load the upper immediate memory address into register \$at (highlighted in green in the *Register* window below).



Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x10010000
\$v0	2	0x00000000
\$v1	3	0x00000000

Figure 3b: Register Window showing the execution of the `lui` instruction.

Note that we have not yet stored variable `a` into register `$s0` as described by the assembly instruction `lw $s0, a`. In the Data Segment window below displays the WORD stored in the memory address `0x10010000` is `0x00000001` or 1 in decimal (zero offset).

Data Segment		
Address	Value (+0)	Value (+4)
0x10010000	0x00000001	0x00000002
0x10010020	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000

Figure 3c: Data segment Window

\$sp	29	0x7ffffeffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400004
hi		0x00000000
lo		0x00000000

Figure 3d: Register Window showing value of \$pc

We are done with the current instruction and so to determine the next instruction, we again look at the `$pc` register value. The value is now `0x00400004`. This can be double checked by viewing the Text Segment window in figure 3a which shows the next instruction highlighted in yellow. We hit F7 to execute the next instruction and which updates the Text Segment window to figure 4a below.

<input type="checkbox"/>	0x00400004	0x8c300000	<code>lw \$16, 0x00000000 (\$1)</code>	
<input type="checkbox"/>	0x00400008	0x3c011001	<code>lui \$1, 0x00001001</code>	9: <code>lw \$s1, b</code>

Figure 4a: Text Segment window showing current instruction & next instruction (in yellow)

The current instruction completes loading variable `a` into register `$s0`. This is done by loading the WORD (i.e. `0x00000001`) at the address stored in register `$at` (i.e. `0x10010000`) with zero offset onto register `$s0`. This is reflected in the Register Window with `$s0` now containing the value `0x00000001` or 1 in decimal. In other words, register `$s0` now contains static variable `a` and thus the assembly instruction `lw $s0, a` is complete.

\$s0	16	0x00000001
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000

Figure 4b: Register Window

The Data Segment window remains unchanged and so we proceed to the next instruction whose address is stored in \$pc.

\$sp	29	0x7ffefffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400008

Figure 4c: Register Window showing value of \$pc

We hit F7 to execute the next instruction and which updates the Text Segment window to figure 5a below.

<input type="checkbox"/>	0x00400008	0x3c011001	lui \$1,0x00001001	9: lw \$s1, b
<input type="checkbox"/>	0x0040000c	0x8c310004	lw \$17,0x00000004(\$1)	

Figure 5a: Text Segment window showing current instruction & next instruction (in yellow)

The previous process is repeated for static variable b. Interestingly, the current instruction is exactly as the first instruction. The value of 0x10010000 is again loaded into register \$at.

\$zero	0	0x00000000
\$at	1	0x10010000
\$v0	2	0x00000000

Figure 5b: Register Window showing the execution of the lui instruction.

The Data Segment window is unchanged and so we proceed to the next instruction whose address is stored in \$pc.

\$sp	29	0x7ffefffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x0040000c

Figure 5c: Register window showing value of \$pc

We hit F7 to execute the next instruction and which updates the Text Segment window to figure 6a below.

<input type="checkbox"/>	0x0040000c	0x8c310004	lw \$t7,0x00000004(\$t1)	←
<input type="checkbox"/>	0x00400010	0x3c011001	lui \$t1,0x00001001	10: lw \$s2, c

Figure 6a: Text Segment window showing current instruction & next instruction (in yellow)

\$zero	0	0x00000000
\$at	1	0x10010000
\$v0	2	0x00000000

Figure 6b: Register window showing value of register \$at

The current instruction completes loading variable `b` into register `$s1`. This is done by loading the WORD from the address stored in register `$at` plus an offset of `0x00000004`. Since the address stored in register `$at` is `0x10010000`, the absolute address becomes `0x10010004`. The WORD stored at this address can be found in the Data Segment window (column 'Value (+4)') below.

Data Segment		
Address	Value (+0)	Value (+4)
0x10010000	0x00000001	0x00000002
0x10010020	0x00000000	0x00000000

Figure 6c: Data Segment window showing value of WORD stored at address `0x10010004`

The WORD stored in the absolute address is `0x00000002` or 2 in decimal. This is the value that will be loaded onto register `$s1` which is reflected in the Register window of figure 6d.

\$s0	16	0x00000001
\$s1	17	0x00000002
\$s2	18	0x00000000
\$s3	19	0x00000000

Figure 6d: Register window showing value of register `$s1`

At this point in the program, register `$s1` now contains static variable `b` and the assembly instruction `lw $s1, b` is complete. We proceed to the next instruction given by the `$pc` register in figure 6e.

\$sp	29	0x7fffeffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400010

Figure 6e: Register window showing value of register \$pc

We hit F7 to execute the next instruction and which updates the Text Segment window to figure 6a below.

0x00400010	0x3c011001	lui \$1,0x00001001	10: lw \$s2, c
0x00400014	0x8c320008	lw \$18,0x00000008(\$1)	

Figure 7a: Text Segment window showing current instruction & next instruction (in yellow)

Again, for the current instruction, just as the first instruction, 0x10010000 is loaded into register \$at as shown in figure 7b.

\$zero	0	0x00000000
\$at	1	0x10010000

Figure 7b: Register window showing the loading of 0x10010000

We proceed to the next instruction given by the \$pc register in figure 7c.

\$sp	29	0x7fffeffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400014

Figure 7c: Register window showing value of \$pc

We hit F7 to execute the next instruction and which updates the Text Segment window to figure 8a below.

0x00400014	0x8c320008	lw \$18,0x00000008(\$1)	
0x00400018	0x3c011001	lui \$1,0x00001001	11: lw \$s3, d

Figure 8a: Text Segment window showing current instruction & next instruction (in yellow)

\$zero	0	0x00000000
\$at	1	0x10010000
\$v0	2	0x00000000

Figure 8b: Register window showing value of register \$at

The current instruction completes loading variable `c` into register `$s2`. This is done by loading the WORD from the address stored in register `$at` plus an offset of `0x00000008`. Since the address stored in register `$at` is `0x10010000`, the absolute address becomes `0x10010008`. The WORD stored at this address can be found in the Data Segment window (column 'Value (+8)') below.

Data Segment			
Address	Value (+0)	Value (+4)	Value (+8)
0x10010000	0x00000001	0x00000002	0x00000003

Figure 8c: Data Segment window showing value of WORD stored at address `0x10010008`

The WORD stored in the absolute address is `0x00000003` or 3 in decimal. This is the value that will be loaded onto register `$s2` which is reflected in the Register window of figure 8d.

<code>\$s0</code>	16	0x00000001
<code>\$s1</code>	17	0x00000002
<code>\$s2</code>	18	0x00000003

Figure 8d: Register window showing value of register `$s2`

At this point in the program, register `$s2` now contains static variable `c` and the assembly instruction `lw $s2, c` is complete. In addition, at this point of we know enough of how these machine instructions work and how the addresses are being loaded into the appropriate registers as described the assembly program. Thus, we know the upper immediate memory address `0x10010000` will be loaded into register `$at` each time we want to load a new variable into a register. Likewise, we know that the offsets increment by +4. So, we know the WORDS stored in the Data Segment window for offsets +8C and +10 are the values `0x00000004` and `0x00000005` respectively.

Data Segment					
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x10010000	0x00000001	0x00000002	0x00000003	0x00000004	0x00000005
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Figure 9a: Data Segment window showing value of WORD stored at addresses `0x10010000` to `0x10010010`

These values will be loaded into registers `$s3` and `$s4` respectively as confirmed in figure 9b and hold the static variables `d` and `e`.

\$s0	16	0x00000001
\$s1	17	0x00000002
\$s2	18	0x00000003
\$s3	19	0x00000004
\$s4	20	0x00000005

Figure 9b: Register window showing value of registers \$s0 through \$s4

The next two instructions have address 0x00400028 and 0x0040002C.

0x00400028	0x02328020	add \$16,\$17,\$18	14: add \$s0, \$s1, \$s2
0x0040002c	0x3c011001	lui \$1,0x00001001	15: sw \$s0, a

Figure 10a: Text Segment window showing the instruction needed to perform $a = b + c$

Address 0x00400028 holds the assembly instruction `add $s0, $s1, $s2` which aims to perform $a = b + c$. In terms of programming, this instruction adds the contents of registers \$s1 and \$s2 (i.e., 2 and 3) and stores the result in register \$s0 (i.e. variable *a*). At this point of the program, register \$s0 contains the value 0x00000005 or 5 in decimal which is expected since $2 + 3$ does indeed equal 5.

\$s0	16	0x00000005
\$s1	17	0x00000002
\$s2	18	0x00000003
\$s3	19	0x00000004
\$s4	20	0x00000005

Figure 10b: Register window showing the updated contents of register \$s0

The next instruction at address 0x0040002C (figure 10a) simply loads the memory address 0x10010000 into register \$at just like the first instruction. The following instruction at address 0x00400030, stores the WORD in register \$s0 back into the memory address 0x10010000. This is reflected in the data segment window of figure 10c.

Data Segment	
Address	Value (+0)
0x10010000	0x00000005

Figure 10c: Data Segment window showing updated value of WORD stored at addresses 0x10010000

We move to the next instruction at address 0x00400034 holds the assembly instruction `sub $s3, $s0, $s4` which aims to perform $d = a - e$.

0x00400034	0x02149822	sub \$19,\$16,\$20	17: sub \$s3, \$s0, \$s4
0x00400038	0x3c011001	lui \$1,0x00001001	18: sw \$s3, d

Figure 11a: Text Segment window showing the instruction needed to perform $d = a - e$

In terms of programming, this instruction subtracts the contents of registers `$s0` and `$s4` (i.e., 5 and 5) and stores the result in register `$s3` (i.e. variable `d`). At this point of the program, register `$s3` contains the value 0x00000000 or 0 in decimal which is expected since $5 - 5$ does indeed equal 0.

\$s0	16	0x00000005
\$s1	17	0x00000002
\$s2	18	0x00000003
\$s3	19	0x00000000
\$s4	20	0x00000005

Figure 11b: Register window showing the updated contents of register `$s3`

The next instruction at address 0x00400038 (figure 11a) simply loads the memory address 0x10010000 into register `$at` just like the first instruction. The following (and last) instruction at address 0x0040003C, stores the WORD in register `$s3` back into the memory address 0x10010000 (+C offset). This is reflected in the data segment window of figure 11c. The program has now successfully terminated.

Data Segment				
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x10010000	0x00000005	0x00000002	0x00000003	0x00000000

Figure 11c: Data Segment window showing updated value of WORD stored at addresses 0x1001000C

2-2_2.asm (Simple Arithmetic)

The following assembly code is similar to the previous program. Five *static* variables `f` through `j` are declared with initialized values. The purpose of this program is to perform three arithmetic operations.

```
.data
f: .word 0
g: .word 50
h: .word 40
i: .word 30
j: .word 20

.text
lw $s0, f
lw $s1, g
lw $s2, h
lw $s3, i
lw $s4, j
# t0 = g + h
add $t0, $s1, $s2
# t1 = i + j
add $t1, $s3, $s4
# f = t0 - t1
sub $s0, $t0, $t1
sw $s0, f
```

The initial Text Segment, Data Segment, and Register windows are shown in figures 12a, 12b, and 12c respectively.

Address	Code	Basic	
0x00400000	0x3c011001	lui \$1,0x00001001	9: lw \$s0, f

Figure 12a: Initial Text Segment window

The initial Text Segment window shows the next instruction highlighted in yellow.

Data Segment					
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x10010000	0x00000000	0x00000032	0x00000028	0x0000001e	0x00000014

Figure 12b: Initial Data Segment window

The initial Data Segment window shows the WORDS stored at each address in memory. It can be seen that variables `f`, `g`, `h`, `i`, and `j` are at address with offsets `+0`, `+4`, `+8`, `+C`, and `+10` with values are 0, 50, 40, 30, and 20 in decimal respectively.

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	

Figure 12c: Initial Register window

In the initial Register window, all register values are initialized to 0 except for the `$gp`, `$sp`, and `$pc` registers. Figure 13a shows the next instruction to be executed. Note that the instructions at addresses 0x00400000 through 0x00400024 load the static variables `f` through `j` into registers `$s0` through `$s4` respectively. The logic behind these instructions was explained in detail in the previous assembly program.

Address	Code	Basic	
0x00400000	0x3c011001	lui \$1,0x00001001	9: lw \$s0, f
0x00400004	0x8c300000	lw \$16,0x00000000(\$1)	
0x00400008	0x3c011001	lui \$1,0x00001001	10: lw \$s1, g
0x0040000c	0x8c310004	lw \$17,0x00000004(\$1)	
0x00400010	0x3c011001	lui \$1,0x00001001	11: lw \$s2, h
0x00400014	0x8c320008	lw \$18,0x00000008(\$1)	
0x00400018	0x3c011001	lui \$1,0x00001001	12: lw \$s3, i
0x0040001c	0x8c33000c	lw \$19,0x0000000c(\$1)	
0x00400020	0x3c011001	lui \$1,0x00001001	13: lw \$s4, j
0x00400024	0x8c340010	lw \$20,0x00000010(\$1)	
0x00400028	0x02324020	add \$8,\$17,\$18	15: add \$t0, \$s1, \$s2

Figure 13a: Text Segment window

Figure 13b shows the register values for those mentioned above.

\$s0	16	0x00000000
\$s1	17	0x00000032
\$s2	18	0x00000028
\$s3	19	0x0000001e
\$s4	20	0x00000014

Figure 13b: Register window showing contents of \$s0 through \$s4

We turn our focus the instruction at address 0x00400028 which holds the assembly instruction `add $t0, $s1, $s2`. This means that the contents of registers `$s1` and `$s2` will be added

and the result is stored in register `$t0`. The value of register `$t0` is updated and shown in figure 13c.

<code>\$t0</code>	8	0x0000005a
<code>\$t1</code>	9	0x00000000

Figure 13c: Register window showing the updated contents of `$t0`

It can be seen that register `$t0` contains the value 0x0000005A or 90 in decimal. This is expected since the contents of registers `$s1` and `$s2` are 50 and 40 respectively. Thus, $50 + 40 = 90$. It should be noted that registers `$t0` through `$t7` are temporary registers and need not be saved. In the next instruction at address 0x0040002C, another addition is performed. This time, the contents of registers `$s3` and `$s4` will be added and the result is stored in register `$t1`.

<code>\$t0</code>	8	0x0000005a
<code>\$t1</code>	9	0x00000032

Figure 13d: Register window showing the updated contents of `$t1`

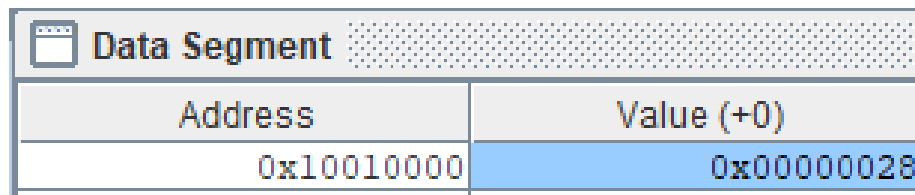
It can be seen that register `$t0` contains the value 0x00000032 or 50 in decimal. This is expected since the contents of registers `$s3` and `$s4` are 30 and 20 respectively. Thus, $30 + 20 = 50$. In the next instruction at address 0x00400034, yet another arithmetic operation is performed. The corresponding assembly instruction is `sub $s0, $t0, $t1`. That is, we are subtracting the contents of the temporary registers `$t0` and `$t1` and storing the result into register `$s0`. Figure 13e shows the updated contents of this register.

<code>\$s0</code>	16	0x00000028
<code>\$s1</code>	17	0x00000032
<code>\$s2</code>	18	0x00000028
<code>\$s3</code>	19	0x0000001e
<code>\$s4</code>	20	0x00000014

Figure 13e: Register window showing the updated contents of `$s0`

This register now contains 0x00000028 or 40 in decimal. This is the expected value since `$t0` and `$t1` contain the values 90 and 50 respectively and so $90 - 50 = 40$. The instruction at address 0x00400034 loads 0x10010000 into the `$at` register. The following (and last) instruction at address 0x00400038 stores the WORD into the memory address 0x10010000 (+0 offset).

Previously, this value was 0 but it is now 40 in decimal. Hence the static variable `f` now holds the value 40 instead of 0.



The image shows a 'Data Segment' window with a table of memory addresses and values. The table has two columns: 'Address' and 'Value (+0)'. The first row shows the address '0x10010000' and the value '0x00000028'. The value is highlighted in blue.

Address	Value (+0)
0x10010000	0x00000028

Figure 13f: Data Segment window showing the updated WORD stored at address 0x1001000

The program terminates.

2-3_1.asm (Arrays)

The following assembly program makes use of two *static* variables *g* and *h* along with a static array *A*. Its purpose is to set $A[8] = 55$, load the value of $A[8]$ into register $\$t0$ and add the contents of registers $\$s2$ and $\$t0$ and store the result in register $\$s1$.

```
.data
g: .word 0
h: .word 22
A: .word 0-100
size: .word 100

.text
# just to set A[8] to 55
li $t1, 55
la $s3, A
sw $t1, 32($s3)
lw $s1, g
lw $s2, h
# loading the value of A[8] into t0
lw $t0, 32($s3)
add $s1, $s2, $t0
sw $s1, g
```

Address	Code	Basic	
0x00400000	0x24090037	addiu \$9,\$0,0x00000037	9: li \$t1, 55
0x00400004	0x3c011001	lui \$1,0x00001001	10: la \$s3, A
0x00400008	0x34330008	ori \$19,\$1,0x00000008	
0x0040000c	0xae690020	sw \$9,0x00000020(\$19)	11: sw \$t1, 32(\$s3)
0x00400010	0x3c011001	lui \$1,0x00001001	12: lw \$s1, g
0x00400014	0x8c310000	lw \$17,0x00000000(\$1)	
0x00400018	0x3c011001	lui \$1,0x00001001	13: lw \$s2, h
0x0040001c	0x8c320004	lw \$18,0x00000004(\$1)	
0x00400020	0x8e680020	lw \$8,0x00000020(\$19)	15: lw \$t0, 32(\$s3)
0x00400024	0x02488820	add \$17,\$18,\$8	16: add \$s1, \$s2, \$t0
0x00400028	0x3c011001	lui \$1,0x00001001	17: sw \$s1, g
0x0040002c	0xac310000	sw \$17,0x00000000(\$1)	

Figure 14a: Text Segment Window showing all program instructions

As in the previous programs, all register values excluding $\$sp$, $\$gp$, and $\$pc$ are initialized to zero. The first instruction at address 0x00400000 is defined by the assembly code `li $t1, 55`. This instruction adds the initial contents of $\$t1$ (i.e. 0) and 55 to store it back into register $\$t1$ via the `addui` command. Register $\$t1$ now contains 55 or 0x00000037 in hexadecimal.

\$t0	8	0x00000000
\$t1	9	0x00000037
\$t2	10	0x00000000

Figure 14b: Register window showing the updated contents of \$t1

In the next instruction at address 0x00400004 simply loads 0x10010000 into register \$at. The following instruction at address 0x00400008 will perform an immediate or (ori) to locate the address of the first element in array A. The register window of figure 14c reflects this as register \$s3 contains 0x10010008, the address of the first element in array A.

\$s2	18	0x00000000
\$s3	19	0x10010008
\$s4	20	0x00000000

Figure 14c: Register window showing the contents of \$s3

The instruction at address 0x0040000C will store the contents of register \$t1 with +32 offset (0x00000020) from address 0x10010008 (i.e. the address of the first element in array A). The absolute address is then 0x10010028. In the Data Segment window of figure 14d, the WORD stored at address 0x10010020 with +8 offset. This value is 0x00000037 or 55 in decimal.

Data Segment			
Address	Value (+0)	Value (+4)	Value (+8)
0x10010000	0x00000000	0x00000016	0x00000000
0x10010020	0x00000000	0x00000000	0x00000037

Figure 14d: Data Segment window showing the WORD stored at address 0x10010028

The instruction at address 0x00400010 once again loads 0x10010000 into register \$at. The next instruction at address 0x00400014 will load 0x00000000 into register \$s1 (with +0 offset). This is reflected in the Register window of figure 14e.

\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x10010008

Figure 14e: Register window showing the contents of \$s1

The instruction at address 0x00400018 once again loads 0x10010000 into register \$at. The next instruction at address 0x0040001C will load 0x00000016 or 22 in decimal into register \$s2 (with +4 offset). This is reflected in the Register window of figure 14f.

\$s1	17	0x00000000
\$s2	18	0x00000016
\$s3	19	0x10010008

Figure 14f: Register window showing the contents of \$s2

The Data Segment window of figure 14g shows the WORDS stored for static variables *g* and *h*. They are each stored at address 0x1001000 with their respective offsets mentioned earlier. It can be seen that these values are 0 and 22 in decimal at offsets +0 and +4 for static variables *g* and *h* respectively.

Data Segment		
Address	Value (+0)	Value (+4)
0x10010000	0x00000000	0x00000016

Figure 14g: Data Segment window showing the WORDS stored at addresses 0x10010000 & 0x10010004

The instruction at address 0x00400020 will load the WORD stored at register \$s3 with +32 offset onto register \$t0 (i.e. the address of *A*[8]). Figure 14h shows the updated contents of register \$t0.

\$t0	8	0x00000037
\$t1	9	0x00000037

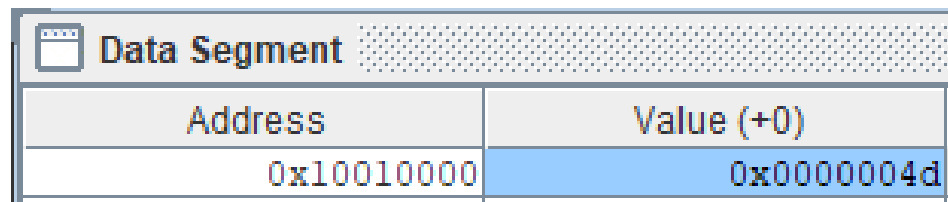
Figure 14h: Register window showing the updated contents of \$t0

The instruction at address 0x00400024 will perform an addition between the contents of registers \$s2 and \$t0 and store the result in register \$s1. Figure 14i shows the updated contents of register \$s1.

\$s1	17	0x0000004d
\$s2	18	0x00000016

Figure 14i: Register window showing the updated contents of \$s1

The value stored in register \$s1 was 0 but is now 0x0000004D or 77 in decimal. This is expected because the contents in registers \$s2 and \$t0 are 22 and 55 respectively and so $22 + 55 = 77$. The instruction at address 0x00400028 will just load 0x10010000 into register \$at. The last instruction at address 0x0040002C stores the WORD at address 0x10010000 with no offset. The value stored is the updated value of static variable *g* (i.e. 77).



The image shows a 'Data Segment' window with a table containing two columns: 'Address' and 'Value (+0)'. The first row of the table has the address '0x10010000' and the value '0x0000004d'. The value cell is highlighted in blue.

Address	Value (+0)
0x10010000	0x0000004d

Figure 14j: Data Segment window showing the updated WORD stored at memory address 0x10010000

The program terminates.

2-3_2.asm (Arrays)

The following assembly program will make use of one *static* variable *h* and a static array *A*. Its purpose is to set $A[8] = 200$ and then perform an addition of the contents of *h* and $A[8]$ and store the result in $A[12]$.

```
.data
h: .word 25
A: .word 0-100
size: .word 100

.text
lw $s2, h
# initializing A[8] to 200
li $t1, 200
la $s3, A
sw $t1, 32($s3)
# A[12] = h + A[8]
lw $t0, 32($s3)
add $t0, $s2, $t0
sw $t0, 48($s3)
```

Text Segment				
Bkpt	Address	Code	Basic	
<input type="checkbox"/>	0x00400000	0x3c011001	lui \$1,0x00001001	7: lw \$s2, h
<input type="checkbox"/>	0x00400004	0x8c320000	lw \$18,0x00000000(\$1)	
<input type="checkbox"/>	0x00400008	0x240900c8	addiu \$9,\$0,0x000000c8	9: li \$t1, 200
<input type="checkbox"/>	0x0040000c	0x3c011001	lui \$1,0x00001001	10: la \$s3, A
<input type="checkbox"/>	0x00400010	0x34330004	ori \$19,\$1,0x00000004	
<input type="checkbox"/>	0x00400014	0xae690020	sw \$9,0x00000020(\$19)	11: sw \$t1, 32(\$s3)
<input type="checkbox"/>	0x00400018	0x8e680020	lw \$8,0x00000020(\$19)	13: lw \$t0, 32(\$s3)
<input type="checkbox"/>	0x0040001c	0x02484020	add \$8,\$18,\$8	14: add \$t0, \$s2, \$t0
<input type="checkbox"/>	0x00400020	0xae680030	sw \$8,0x00000030(\$19)	15: sw \$t0, 48(\$s3)

Figure 15a: Text Segment Window showing all program instructions

As in the previous programs, all register values excluding $\$sp$, $\$gp$, and $\$pc$ are initialized to zero. The first instruction at address 0x00400000 will load 0x10010000 to register $\$at$.

$\$zero$	0	0x00000000
$\$at$	1	0x10010000

Figure 15b: Register window showing the contents of register $\$at$ after first instruction

The next instruction at address 0x00400004 will load the value 0x00000019 into register $\$s2$. That is, register $\$s2$ now contains the value static variable *h*.

\$s1	17	0x00000000
\$s2	18	0x00000019

Figure 15c: Register window showing the contents of register \$s2

In the Data Segment window, the WORD stored in the address 0x10010000 (+0 offset) is the static variable h.

Data Segment	
Address	Value (+0)
0x10010000	0x00000019

Figure 15d: Data Segment window showing the location of static variable h in memory

The next instruction at address 0x00400008 will store 0x000000C8 or 200 in decimal into temporary register \$t1 by adding that register's initial contents (i.e. 0) and 200.

\$t1	9	0x000000c8
\$t2	10	0x00000000

Figure 15e: Register window showing the contents of register \$t1

The instructions at addresses 0x0040000C and 0x00400010 locate the address of A[0]. At this point of the program, register \$s3 contains the address of A[0].

\$s3	19	0x10010004
\$s4	20	0x00000000

Figure 15f: Register window showing the contents of register \$s3

The instruction at address 0x00400014 stores the WORD in register \$t1 at address 0x10010004 with +20 offset. Alternatively, this address can be represented by 0x10010020 with +4 offset.

Data Segment		
Address	Value (+0)	Value (+4)
0x10010000	0x00000019	0x00000000
0x10010020	0x00000000	0x000000c8

Figure 15g:

The instruction at address 0x00400018 loads the value of $A[8]$ into the temporary register $\$t0$.

$\$t0$	8	0x000000c8
$\$t1$	9	0x000000c8

Figure 15h: Register window showing the contents of register $\$t0$

The following instruction at address 0x0040001C performs an addition of the contents of registers $\$t0$ and $\$s2$ and stores the result back into register $\$t0$. The value stored in register $\$t0$ is now updated to contain 0x000000E1 or 225 in decimal.

$\$t0$	8	0x000000e1
$\$t1$	9	0x000000c8

Figure 15i: Register window showing the updated contents of register $\$t0$

The final instruction at address 0x00400020 will store the WORD in register $\$t0$ at address 0x10010004 with +30 offset. Alternatively, this address can be represented by 0x10010020 with +14 offset. This is reflected in the Data Segment window of figure 15j.

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)
0x10010000	0x00000019	0x00000000	0xffffffff9c	0x00000064	0x00000000	0x00000000
0x10010020	0x00000000	0x000000c8	0x00000000	0x00000000	0x00000000	0x000000e1

Figure 15j: Data Segment window

The program terminates.

2-5_2.asm (Arrays)

The following assembly program will make use of one *static* variable *h* and a static array *A*. Its purpose is to perform the C equivalent of $A[300] = h + A[300]$.

```
.data
h: .word 20
A: .word 0-400
size: .word 400

.text
la $t1, A
lw $s2, h
# initializing A[300] to 13
li $t2, 13
sw $t2, 1200($t1)
lw $t0, 1200($t1)
add $t0, $s2, $t0
sw $t0, 1200($t1)
```

Text Segment				
Bkpt	Address	Code	Basic	
<input type="checkbox"/>	0x00400000	0x3c011001	lui \$1,0x00001001	7: la \$t1, A
<input type="checkbox"/>	0x00400004	0x34290004	ori \$9,\$1,0x00000004	
<input type="checkbox"/>	0x00400008	0x3c011001	lui \$1,0x00001001	8: lw \$s2, h
<input type="checkbox"/>	0x0040000c	0x8c320000	lw \$18,0x00000000(\$1)	
<input type="checkbox"/>	0x00400010	0x240a000d	addiu \$10,\$0,0x0000...	10: li \$t2, 13
<input type="checkbox"/>	0x00400014	0xad2a04b0	sw \$10,0x000004b0(\$9)	11: sw \$t2, 1200(\$t1)
<input type="checkbox"/>	0x00400018	0x8d2804b0	lw \$8,0x000004b0(\$9)	12: lw \$t0, 1200(\$t1)
<input type="checkbox"/>	0x0040001c	0x02484020	add \$8,\$18,\$8	13: add \$t0, \$s2, \$t0
<input type="checkbox"/>	0x00400020	0xad2804b0	sw \$8,0x000004b0(\$9)	14: sw \$t0, 1200(\$t1)

Figure 16a: Text Segment Window showing all program instructions

As in the previous programs, all register values excluding *\$sp*, *\$gp*, and *\$pc* are initialized to zero. The first instruction at address 0x00400000 will load 0x10010000 to register *\$at*.

\$at	1	0x10010000
\$v0	2	0x00000000

Figure 16b: Register window showing the contents of register *\$at* after first instruction

The next instruction at address 0x00400004 loads the value of register \$at +4 into register \$t1 via the `ori` command.

\$t0	8	0x00000000
\$t1	9	0x10010004
\$t2	10	0x00000000

Figure 16c: Register window showing the contents of register \$t1

The instruction at address 0x00400008 again loads 0x10010000 into register \$at. The next instruction at address 0x0040000C loads 0x00000014 or 20 in decimal (i.e. static variable h) onto register \$s3.

\$s1	17	0x00000000
\$s2	18	0x00000014
\$s3	19	0x00000000

Figure 16d: Register window showing the contents of register \$s2

The instruction at address 0x00400010 will load the value 0x0000000D or 13 in decimal onto register \$t2 by adding 0 and 13 via the `addiu` command.

\$t2	10	0x0000000d
\$t3	11	0x00000000

Figure 16e: Register window showing the contents of register \$t2

The instruction at address 0x00400014 will store the WORD in register \$t2 to memory address 0x100104B4. Alternatively, this address can be represented by 0x10014A0 with +14 offset.

Data Segment						
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)
0x10010400	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010420	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010440	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010460	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010480	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100104a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x0000000d

Figure 16f: Data Segment window showing the WORD stored at address 0x100104B4

At this point of the program, $A[300] = 13$. The instruction at address $0x00400018$ will load the stored WORD from memory address $0x100104B4$ (previous instruction) onto register $\$t0$.

$\$t0$	8	$0x0000000d$
$\$t1$	9	$0x10010004$
$\$t2$	10	$0x0000000d$

Figure 16f: Register window showing the contents of register $\$t0$

The instruction at address $0x0040001C$ will perform an addition of the contents of registers $\$s2$ and $\$t0$ and store the result back into register $\$t0$.

$\$t0$	8	$0x00000021$
$\$t1$	9	$0x10010004$
$\$t2$	10	$0x0000000d$

Figure 16g: Register window showing the updated contents of register $\$t0$

The value now stored in register $\$t0$ is $0x00000021$ or 33 in decimal. This is expected since the contents in registers $\$s2$ and $\$t0$ are 20 and 13 respectively. Thus, $20 + 13 = 33$. The last instruction stores the WORD in register $\$t0$ into memory address $0x100104B4$ overwriting any previous value.

Data Segment						
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)
$0x10010400$	$0x00000000$	$0x00000000$	$0x00000000$	$0x00000000$	$0x00000000$	$0x00000000$
$0x10010420$	$0x00000000$	$0x00000000$	$0x00000000$	$0x00000000$	$0x00000000$	$0x00000000$
$0x10010440$	$0x00000000$	$0x00000000$	$0x00000000$	$0x00000000$	$0x00000000$	$0x00000000$
$0x10010460$	$0x00000000$	$0x00000000$	$0x00000000$	$0x00000000$	$0x00000000$	$0x00000000$
$0x10010480$	$0x00000000$	$0x00000000$	$0x00000000$	$0x00000000$	$0x00000000$	$0x00000000$
$0x100104a0$	$0x00000000$	$0x00000000$	$0x00000000$	$0x00000000$	$0x00000000$	$0x00000021$

Figure 16f: Data Segment window showing the updated WORD stored at address $0x100104B4$

At this point of the program, the value of $A[300]$ is 33. The last instruction has been executed and the program terminates.

2-6_1.asm (Bitwise Operations)

The following assembly program performs a left shift followed by three bitwise operations: AND, OR, and NOT.

```
# left shift
li $s0, 9
sll $t2, $s0, 4
# AND
li $t2, 0xdc0
li $t1, 0x3c00
and $t0, $t1, $t2
# OR
or $t0, $t1, $t2
# NOTs
li $t3, 0
nor $t0, $t1, $t3
```

Text Segment				
Bkpt	Address	Code	Basic	
<input type="checkbox"/>	0x00400000	0x24100009	addiu \$16,\$0,0x0000...	2: li \$s0, 9
<input type="checkbox"/>	0x00400004	0x00105100	sll \$10,\$16,0x00000004	3: sll \$t2, \$s0, 4
<input type="checkbox"/>	0x00400008	0x240a0dc0	addiu \$10,\$0,0x0000...	5: li \$t2, 0xdc0
<input type="checkbox"/>	0x0040000c	0x24093c00	addiu \$9,\$0,0x00003c00	6: li \$t1, 0x3c00
<input type="checkbox"/>	0x00400010	0x012a4024	and \$8,\$9,\$10	7: and \$t0, \$t1, \$t2
<input type="checkbox"/>	0x00400014	0x012a4025	or \$8,\$9,\$10	9: or \$t0, \$t1, \$t2
<input type="checkbox"/>	0x00400018	0x240b0000	addiu \$11,\$0,0x0000...	11: li \$t3, 0
<input type="checkbox"/>	0x0040001c	0x012b4027	nor \$8,\$9,\$11	12: nor \$t0, \$t1, \$t3

Figure 17a: Text Segment window of 2-6_1.asm

The first instruction at address 0x00400000 loads the value 0x00000009 into register `$s0`. It does so by adding the initial contents of `$s0` (i.e. 0) and 9 to store it back into register `$s0` via the `addui` command.

<code>\$s0</code>	16	0x00000009
<code>\$s1</code>	17	0x00000000

Figure : Register window showing the updated contents of register `$s0`.

The instruction at address 0x00400004 performs a *left shift* via the `sll` command. The left shift is applied to the value stored in register `$s0` (i.e. 9). That is, 0x00000009 is shifted 4 binary bits to the left (recall that 1 hexadecimal digit = 4 binary bits) resulting in 0x00000090 which is stored in register `$t2`.

\$t2	10	0x00000090
\$t3	11	0x00000000

Figure 17b: Register window showing the updated contents of register \$t2.

The instructions at addresses 0x00400008 and 0x0040000C are similar to the first instruction. They load the values 0x00000DC0 and 0x00003C00 into temporary registers \$t1 and \$t2 respectively.

\$t1	9	0x00003c00
\$t2	10	0x00000dc0

Figure 17c: Register window showing the updated contents of registers \$t1 and \$t2.

The next instruction at address 0x00400010 performs a bitwise AND operation on the contents of registers \$t1 and \$t2 and stores the result onto register \$t0.

\$t0	8	0x00000c00
\$t1	9	0x00003c00
\$t2	10	0x00000dc0

Figure 17d: Register window showing the updated contents of registers \$t0 after bitwise AND operation

The next instruction at address 0x00400014 performs a bitwise OR on the contents of registers \$t1 and \$t2 and stores the result onto register \$t0.

\$t0	8	0x00003dc0
\$t1	9	0x00003c00
\$t2	10	0x00000dc0

Figure 17e: Register window showing the updated contents of register \$t0 after bitwise OR operation

The next instruction at address 0x00400018 loads 0 onto temporary register \$t3.

\$t0	8	0x00003dc0
\$t1	9	0x00003c00
\$t2	10	0x00000dc0
\$t3	11	0x00000000

Figure 17f: Register window showing the updated contents of register \$t0

The last instruction at address 0x0040001C performs a bitwise NOR operation on the contents of registers \$t1 and \$t3 and stores the result onto register \$t0.

\$t0	8	0xffffc3ff
\$t1	9	0x00003c00
\$t2	10	0x00000dc0
\$t3	11	0x00000000

Figure 17: Register window showing the updated contents of register \$t0 after bitwise NOR operation

The program terminates.

If_Else.asm

The following program makes use of four static variables a through d and demonstrates the concept of the If-Else statement. Here, if $a = d$, then $a = b + c$. Else, $a = b - c$.

```
.data
a: .word 0
b: .word 100
c: .word 80
d: .word 60

.text
lw $s0, a
lw $s1, b
lw $s2, c
lw $s3, d
bne $s0, $s3, Else
# if a == d, then
add $s0, $s1, $s2 # a = b+c
j Exit
Else:
# else
sub $s0, $s1, $s2 # a = b-c
Exit:
sw $s0, a
```

Text Segment				
Bkpt	Address	Code	Basic	
<input type="checkbox"/>	0x00400000	0x3c011001	lui \$1,0x00001001	8: lw \$s0, a
<input type="checkbox"/>	0x00400004	0x8c300000	lw \$16,0x00000000(\$1)	
<input type="checkbox"/>	0x00400008	0x3c011001	lui \$1,0x00001001	9: lw \$s1, b
<input type="checkbox"/>	0x0040000c	0x8c310004	lw \$17,0x00000004(\$1)	
<input type="checkbox"/>	0x00400010	0x3c011001	lui \$1,0x00001001	10: lw \$s2, c
<input type="checkbox"/>	0x00400014	0x8c320008	lw \$18,0x00000008(\$1)	
<input type="checkbox"/>	0x00400018	0x3c011001	lui \$1,0x00001001	11: lw \$s3, d
<input type="checkbox"/>	0x0040001c	0x8c33000c	lw \$19,0x0000000c(\$1)	
<input type="checkbox"/>	0x00400020	0x16130002	bne \$16,\$19,0x00000002	12: bne \$s0, \$s3, Else
<input type="checkbox"/>	0x00400024	0x02328020	add \$16,\$17,\$18	14: add \$s0, \$s1, \$s2
<input type="checkbox"/>	0x00400028	0x0810000c	j 0x00400030	15: j Exit
<input type="checkbox"/>	0x0040002c	0x02328022	sub \$16,\$17,\$18	18: sub \$s0, \$s1, \$s2
<input type="checkbox"/>	0x00400030	0x3c011001	lui \$1,0x00001001	20: sw \$s0, a
<input type="checkbox"/>	0x00400034	0xac300000	sw \$16,0x00000000(\$1)	

Figure 18a: Text Segment window of IF_ELSE.asm showing all instructions

The instructions at addresses 0x00400000 through 0x0040001C load the static variables a through d to registers \$s0 through \$s3 respectively.

\$s0	16	0x00000000
\$s1	17	0x00000064
\$s2	18	0x00000050
\$s3	19	0x0000003c

Figure 18b: Register window showing the contents of registers \$s0 through \$s3

The next instruction at address 0x00400020 compares the contents of \$s0 and \$s3 (i.e. compares if a=d). If true, the next instruction to be executed will be at address 0x00400024. If not, the next instruction to be executed will be at address 0x0040002C.

0x00400020	0x16130002	bne \$16,\$19,0x00000002	12: bne \$s0, \$s3, Else
0x00400024	0x02320020	add \$16,\$17,\$18	14: add \$s0, \$s1, \$s2
0x00400028	0x00810000c	j 0x00400030	15: j Exit
0x0040002c	0x02320022	sub \$16,\$17,\$18	18: sub \$s0, \$s1, \$s2

Figure 18c: Text Segment window showing the next instruction after execution of bne

After executing the instruction at address 0x00400020, it can be seen that the next instruction is at address 0x0040002C. This is expected because the contents of \$s0 and \$s3 are 100 and 60 respectively and are thus not equal. The next instruction will perform a subtraction on the contents of registers \$s1 and \$s2 and store the result back into register \$s0.

\$s0	16	0x00000014
\$s1	17	0x00000064
\$s2	18	0x00000050
\$s3	19	0x0000003c

Figure 18d: Register window showing the updated contents of register \$s0

The value now stored in register \$s0 is 0x00000014 or 20 in decimal which is the expected result. The last two instructions at addresses 0x00400030 and 0x00400034 will store the contents of register \$s0 onto the effective memory WORD address 0x10010000 (zero offset).

Data Segment	
Address	Value (+0)
0x10010000	0x00000014

Figure 18e: Data Segment window showing the storing of \$s0 in memory

The program terminates.

WhileLoop.asm

The following program will make use of two static variable `a` and `b` to demonstrate a while loop. This value of `a` will just be incremented by one so long as the value is less than a specified number defined by `b`.

```
.data
a: .word 0
b: .word 10
.text
lw $s0, a
lw $s1, b

WhileLoop:
    bgt $s0, $s1, Exit

    addi $s0, $s0, 1 #a++

    j WhileLoop
Exit:

sw $s0, a
```

Text Segment				
Bkpt	Address	Code	Basic	
<input type="checkbox"/>	0x00400000	0x3c011001	lui \$1,0x00001001	5: lw \$s0, a
<input type="checkbox"/>	0x00400004	0x8c300000	lw \$16,0x00000000(\$1)	
<input type="checkbox"/>	0x00400008	0x3c011001	lui \$1,0x00001001	6: lw \$s1, b
<input type="checkbox"/>	0x0040000c	0x8c310004	lw \$17,0x00000004(\$1)	
<input type="checkbox"/>	0x00400010	0x0230082a	slt \$1,\$17,\$16	9: bgt \$s0, \$s1, Exit
<input type="checkbox"/>	0x00400014	0x14200002	bne \$1,\$0,0x00000002	
<input type="checkbox"/>	0x00400018	0x22100001	addi \$16,\$16,0x0000...	11: addi \$s0, \$s0, 1 #i++
<input type="checkbox"/>	0x0040001c	0x08100004	j 0x00400010	13: j WhileLoop
<input type="checkbox"/>	0x00400020	0x3c011001	lui \$1,0x00001001	16: sw \$s0, a
<input type="checkbox"/>	0x00400024	0xac300000	sw \$16,0x00000000(\$1)	

Figure 19a: Text Segment of WhileLoop.asm showing all instructions

The instructions at addresses 0x00400000 to 0x0040000C will load static variables `a` and `b` to registers `$s0` and `$s1` respectively.

\$s0	16	0x00000000
\$s1	17	0x0000000a

Figure 19b: Register window showing the contents of registers `$s0` and `$s1`

The next instruction at address 0x00400010, will determine if $\$s0 \geq \$s1$. Since the value stored in $\$s0$ is 0, the statement is false and so we move to the next instruction at address 0x00400018 where 1 is added to the contents of register $\$s0$.

$\$s0$	16	0x00000001
$\$s1$	17	0x0000000a

Figure 19c: Register window showing the updated contents of register $\$s0$ after 1 iteration of while loop

The contents of register $\$s0$ is now 1. The next instruction at address 0x0040001C will execute a jump back into the start of the while loop and repeat the above process. In the second to last iteration of the while loop, the contents of register $\$s0$ is 11 in decimal.

$\$s0$	16	0x0000000b
$\$s1$	17	0x0000000a

Figure 19e: Register window showing the updated contents of register $\$s0$ after final iteration of while loop

At the final iteration, the while loop will not be executed since at this point, $\$s0 > \$s1$. The program will then jump to the instruction at address 0x00400020. This instruction, followed by the next, will store the contents of register $\$s0$ onto the effective memory WORD address 0x10010000 (zero offset).

Data Segment	
Address	Value (+0)
0x10010000	0x0000000b

Figure 19f: Data Segment window showing the storing of $\$s0$ in memory

Local.asm

In all previous examples, the variables used were said to be *static* variables. In terms of MIPS, static variables were variables declared in the `.data` section. They were stored in memory addresses outside the stack. Local variables on the other hand, require memory allocation on the stack and storing the values in the registers to the stack. The following program demonstrates the functionality of local variables by swapping `x` and `y` which are locally declared.

```
.text
addi $s0, $s0, 5
addi $s1, $s1, 10
addi $t0, $t0, 0
addi $sp, $sp, -8 # adjust stack pointer (push)

sw $s0, 4($sp)
sw $s1, 0($sp)

add $t0, $zero, $s0 # t0 = s0
add $s0, $zero, $s1 # s0 = s1
add $s1, $zero, $t0 # s1 = s0

sw $s0, 4($sp)
sw $s1, 0($sp)

addi $sp, $sp, 8 # restore stack pointer (pop)
```

Text Segment				
Bkpt	Address	Code	Basic	
<input type="checkbox"/>	0x00400000	0x22100005	addi \$16,\$16,0x0000...	2: addi \$s0, \$s0, 5
<input type="checkbox"/>	0x00400004	0x2231000a	addi \$17,\$17,0x0000...	3: addi \$s1, \$s1, 10
<input type="checkbox"/>	0x00400008	0x21080000	addi \$8,\$8,0x00000000	4: addi \$t0, \$t0, 0
<input type="checkbox"/>	0x0040000c	0x23bdfff8	addi \$29,\$29,0xffff...	5: addi \$sp, \$sp, -8 # adjust stack pointer (push)
<input type="checkbox"/>	0x00400010	0xafb00004	sw \$16,0x00000004(\$29)	7: sw \$s0, 4(\$sp)
<input type="checkbox"/>	0x00400014	0xafb10000	sw \$17,0x00000000(\$29)	8: sw \$s1, 0(\$sp)
<input type="checkbox"/>	0x00400018	0x00104020	add \$8,\$0,\$16	10: add \$t0, \$zero, \$s0 # t0 = s0
<input type="checkbox"/>	0x0040001c	0x00118020	add \$16,\$0,\$17	11: add \$s0, \$zero, \$s1 # s0 = s1
<input type="checkbox"/>	0x00400020	0x00088820	add \$17,\$0,\$8	12: add \$s1, \$zero, \$t0 # s1 = s0
<input type="checkbox"/>	0x00400024	0xafb00004	sw \$16,0x00000004(\$29)	14: sw \$s0, 4(\$sp)
<input type="checkbox"/>	0x00400028	0xafb10000	sw \$17,0x00000000(\$29)	15: sw \$s1, 0(\$sp)
<input type="checkbox"/>	0x0040002c	0x23bd0008	addi \$29,\$29,0x0000...	17: addi \$sp, \$sp, 8 # restore stack pointer (pop)

Figure 20a: Text Segment of Local.asm showing all instructions

The instructions at addresses 0x00400000 through 0x00400008 store the local variables `s0`, `s1`, and `t0` to registers `$s0`, `$s2`, and `$t0` respectively.

\$t0	8	0x00000000
\$s0	16	0x00000005
\$s1	17	0x0000000a

Figure 20b: Register window showing the contents of registers `$s0`, `$s1` and `$t0`

The instruction at address 0x0040000C adjusts the stack pointer.

\$gp	28	0x10008000
\$sp	29	0x7fffff4
\$fp	30	0x00000000

Figure 20c: Showing the contents of \$sp

The instructions at addresses 0x00400010 and 0x00400014 store the contents of registers \$s0 and \$s1 onto the effective memory WORD address 0x7ffffe0 with offsets +18 and +14 respectively.

Data Segment							
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)
0x7ffffe0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x0000000a	0x00000005

Figure 20d: Data Segment window showing the storing of \$s0 and \$s1 in memory

The instructions at addresses 0x00400018 to 0x0040002C will perform the swap by first copying the contents of register \$s0 onto the temporary register \$t0. The contents of register \$s1 are then copied to register \$s0. Lastly, the contents of register \$t0 are copied to register \$s1.

\$s0	16	0x0000000a
\$s1	17	0x00000005

Figure 20e: Register window showing the updated contents of registers \$s0 and \$s1 after swapping

The next two instructions at addresses 0x00400024 and 0x00400028 store the updated contents of registers \$s0 and \$s1 onto the effective memory WORD address 0x7ffffe0 with offsets +18 and +14 respectively.

Data Segment							
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)
0x7ffffe0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000005	0x0000000a

Figure 20f: Data Segment window showing the storing of \$s0 and \$s1 in memory

The last instruction at address 0x0040002C restores the stack pointer and the program terminates.

\$gp	28	0x10008000
\$sp	29	0x7fffffc
\$fp	30	0x00000000

Figure 20g: Showing the contents of \$sp

Natural_Generator.asm

The following program utilized both one *local* variable *a* and *static* variable *b* to perform simple arithmetic. For the first time, we will be dealing with negative numbers.

```
.data
b: .word -1 #static variable declaration

.text
lw $s0, b # b = -1
addi $s1, $s1, 1 # a = 1
add $s0, $s0, 1 # b += 1
add $s0, $s1, $s0 # a+b
```

Text Segment				
Bkpt	Address	Code	Basic	
<input type="checkbox"/>	0x00400000	0x3c011001	lui \$1,0x00001001	5: lw \$s0, b # b = -1
<input type="checkbox"/>	0x00400004	0x8c300000	lw \$16,0x00000000(\$1)	
<input type="checkbox"/>	0x00400008	0x22310001	addi \$17,\$17,0x00000001	6: addi \$s1, \$s1, 1 # a = 1
<input type="checkbox"/>	0x0040000c	0x23bdfffc	addi \$29,\$29,0xffffffffc	7: addi \$sp, \$sp, -4 # adjust stack pointer (push)
<input type="checkbox"/>	0x00400010	0x22100001	addi \$16,\$16,0x00000001	8: add \$s0, \$s0, 1 # b += 1
<input type="checkbox"/>	0x00400014	0x02308020	add \$16,\$17,\$16	9: add \$s0, \$s1, \$s0 # a+b
<input type="checkbox"/>	0x00400018	0x23bd0004	addi \$29,\$29,0x00000004	10: addi \$sp, \$sp, 4 # restore stack pointer (pop)

Figure 21a: Text Segment of Natural_Generator.asm showing all instructions

The first two instructions at addresses 0x00400000 and 0x00400004 loads the *static* variable *b* onto register *\$s0*. The instruction at address 0x00400008 loads the *local* variable *a* onto register *\$s1*.

\$s0	16	0xffffffff
\$s1	17	0x00000001

Figure 21b: Register window showing the contents of registers *\$s0* and *\$s1*

Note that the value stored in register *\$s0* is the signed hexadecimal representation of -1. The instruction in address 0x0040000C adjusts the stack. The instruction address 0x00400010 adds 1 to the contents of register *\$s0* (i.e. *b*+1).

\$s0	16	0x00000000
\$s1	17	0x00000001

Figure 21c: Register window showing the updated contents of registers *\$s0*

The instruction at address 0x00400014 adds the contents of registers `$s0` and `$s1` and stores the result back into `$s0`.

<code>\$s0</code>	16	0x00000001
<code>\$s1</code>	17	0x00000001

Figure 21d: Register window showing the updated contents of registers `$s0` (result of `a+b`)

The final instruction restores the stack pointer and the program terminates.

<code>\$sp</code>	29	0x7ffffeffc
<code>\$sp</code>	29	0x7ffffeff8
<code>\$sp</code>	29	0x7ffffeffc

Figure 21e: Contents of stack pointer throughout the program

Part II: Intel X86 on MS Visual Studio (32-Bit)

Topics Covered: Static/Local variables, Arrays, Bitwise Operations, Condition Statements, Loops

Debugging a C program in MS Visual Studio

To explore the Intel X86 ISA, we will utilize Microsoft Visual Studio to debug the previous examples above. I will specifically be using *MS Visual Studio 2017 Community Version*. To debug a C program in MS Visual Studio, we must first create a *Console Application* project.

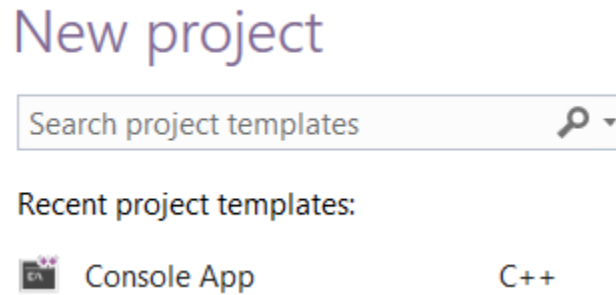


Figure 22a: Creating a new project in MS Visual Studio

For an existing project, we must open the solution (.sln) file and build it. This can be done by hitting CTRL-SHIFT-B.

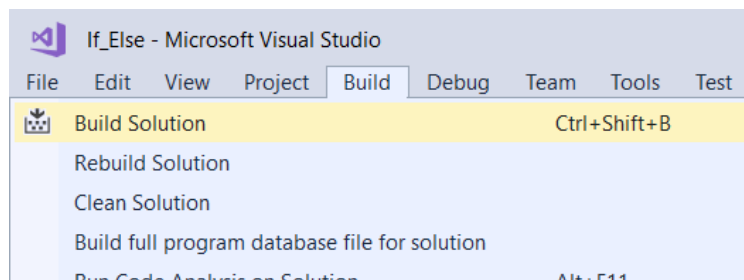


Figure 22b: Building solution file

Next, we initialize the debugger by going into Debug >> Step Into. Alternatively, hit F11.

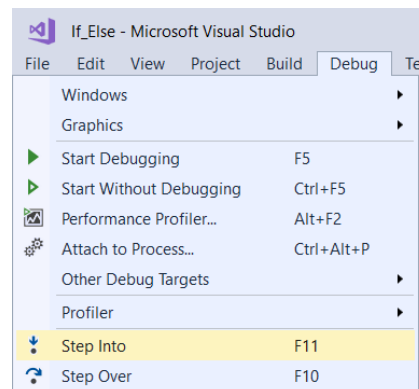


Figure 22c: Initializing the debugger

The Disassembly, Register, and Memory windows should appear. If not, go to Debug >> Windows >> Disassembly (or Registers, or Memory 1). Alternatively, CTRL+ALT+D, CTRL+ALT+G, and CTRL+ALT+M will bring up the Disassembly, Register, and Memory windows respectively.

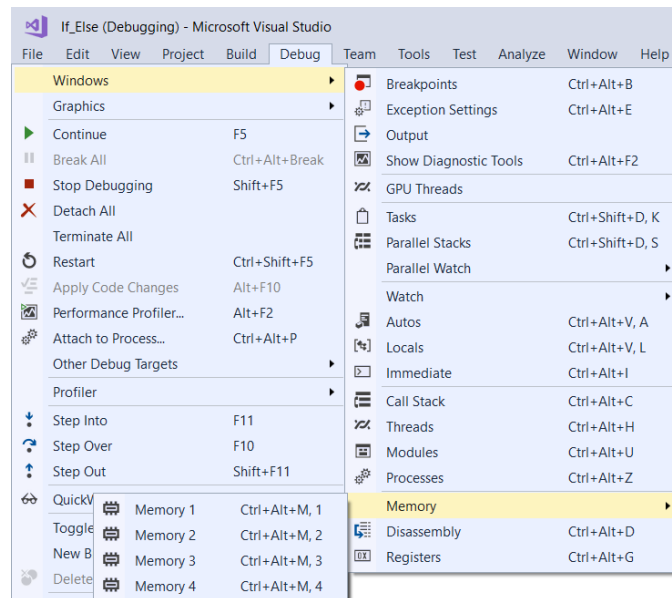


Figure 22d: Displaying the Disassembly, Register, and Memory Window.

To step into the next instruction, simply hit F10.

2-2_1.c (Simple Arithmetic)

```

void main() {
    static int a = 1;
    static int b = 2;
    static int c = 3;
    static int d = 4;
    static int e = 5;
    a = b + c;
    d = a - e;
}

static int a = 1;
static int b = 2;
static int c = 3;
static int d = 4;
static int e = 5;
a = b + c;
00651718 mov     eax,dword ptr [b (0659004h)]
0065171D add     eax,dword ptr [c (0659008h)]
00651723 mov     dword ptr [a (0659000h)],eax
        d = a - e;
00651728 mov     eax,dword ptr [a (0659000h)]
0065172D sub     eax,dword ptr [e (0659010h)]
00651733 mov     dword ptr [d (065900Ch)],eax
}

```

Registers	
EAX = 78BF4750	EBX = 00CB1000
ECX = 00000001	EDX = 00659594
ESI = 00651320	EDI = 00651320
EIP = 006516F0	ESP = 00BBFA54
EBP = 00BBFA64	EFL = 00000202

Memory 1	
0x00BF9004	02 00 00 00
0x00BF9008	03 00 00 00
0x00BF900C	04 00 00 00
0x00BF9010	05 00 00 00
0x00BF9014	00 00 00 00
0x00BF9018	01 00 00 00
0x00BF901C	01 00 00 00
0x00BF9020	01 00 00 00
0x00BF9024	01 00 00 00
0x00BF9028	01 00 00 00
0x00BF902C	00 00 00 00
0x00BF9030	ff ff ff ff
0x00BF9034	01 00 00 00
0x00BF9038	71 bf ac 96
0x00BF903C	8e 40 53 69
0x00BF9040	00 00 00 00
0x00BF9044	2f 00 00 00
0x00BF9048	01 00 00 00
0x00BF904C	00 00 00 00

Figure 23a: Disassembly, Register, & Memory windows of 2-2_1.c

The instruction at address 0x00651718 will copy the value stored at the memory address 0x00BF9004 (i.e. static variable b) onto register EAX.

Registers	
EAX = 00000002	
EBX = 008F8000	
ECX = 00BFB000	
EDX = 00000001	

Figure 23b: Register window showing the contents of register EAX

The instruction at address 0x0065171D will then add the value stored at the memory address 0x00BF9008 (i.e. static variable *c*) to the current contents of register *EAX*. This is the C equivalent of $b+c$.

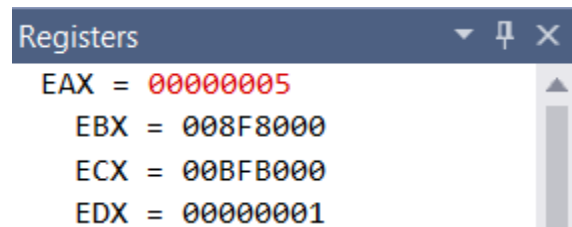


Figure 23c: Register window showing the updated contents of register *EAX*

The value stored at register *EAX* is now 5 in decimal. This is the expected result in the values of static variables *b* and *c* are 2 and 3 respectively. Thus, $2 + 3 = 5$. This result is the value of static variable *a*. The next instruction at address 0x00651723 will copy result of the addition into the memory address 0x00BF9000 (i.e. the memory location of *a*).

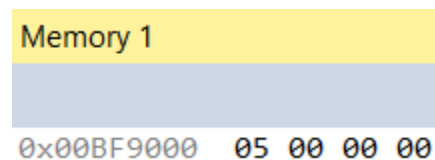


Figure 23d: Memory window showing the location and value of *a*

The next instruction at address 0x00651728 will copy the value stored at the memory address 0x00BF9000 (i.e. static variable *a*) onto register *EAX*. The next instruction at address 0x0065172D will perform a subtraction on the contents of register *EAX* and the value of static variable *e* and store the result back into register *EAX*.

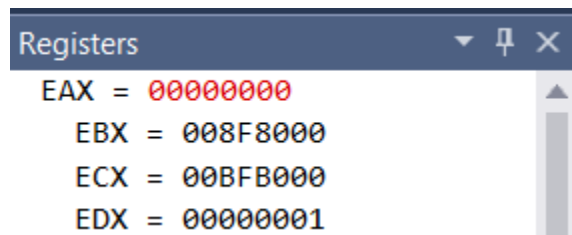


Figure 23e: Register window showing the updated contents of register *EAX*

The value stored at register *EAX* is now 0 in decimal. This is the expected result in the values of static variables *a* and *e* are both 5. Thus, $5 - 5 = 0$. This result is the value of static variable *d*.

The next instruction at address 0x00651733 will copy result of the subtraction into the memory address 0x00BF900C (i.e. the memory location of d).

Memory 1				
0x00BF9000	05	00	00	00
0x00BF9004	02	00	00	00
0x00BF9008	03	00	00	00
0x00BF900C	00	00	00	00
0x00BF9010	05	00	00	00

Figure 23f: Memory window showing the location and value of d

The last sequence of instructions perform some housekeeping. The registers are popped, the memory previously allocated on the stack is deallocated, and the base pointer is popped.

2-3_1.c (Arrays)

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

```
static int g = 0;
static int h = 22;
static int A[100];
A[8] = 55;
00361718 mov     eax,4
0036171D shl     eax,3
00361720 mov     dword ptr A (0369148h)[eax],37h
    g = h + A[8];
0036172A mov     eax,4
0036172F shl     eax,3
00361732 mov     ecx,dword ptr [h (0369000h)]
00361738 add     ecx,dword ptr A (0369148h)[eax]
0036173E mov     dword ptr [g (0369140h)],ecx
}
```

Figure 24a: Disassembly Window of 2-3_1.c

The first three instructions at addresses 0x00361718 to 0x0036171D will compute the offset from the base address of the address to find `A[8]`. First, the value 4 is copied to register EAX and is then shifted 3 bits to the left to yield 0x00000020 or 32 in decimal.

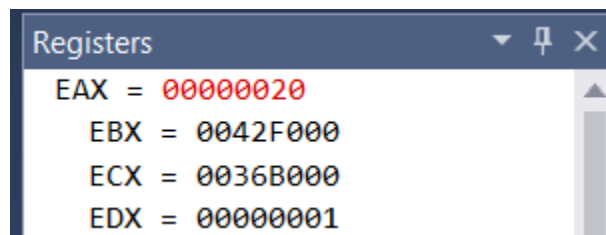


Figure 24b: Contents of register EAX after first two instructions

This value (32) will be the offset to find `A[8]`. The assembly code tells us that the base address of `A` is 0x00369148. Using this base address, we add the value stored in the EAX register to get 0x00369168. This is the memory location of `A[8]`. The value 0x37 or 55 in decimal, is then stored in this memory address.

Figure 24c: Memory window showing the location and value of $A[8]$

At this point of the program, the three instructions performed were equivalent to $A[8] = 55$ in C. The next two instructions at addresses 0x0036172A and 0x0036172F repeat the previous process to find $A[8]$. The instruction at address 0x00361732 will copy the value stored at memory address 0x00369000 onto register ECX.

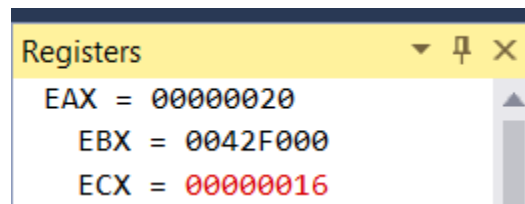
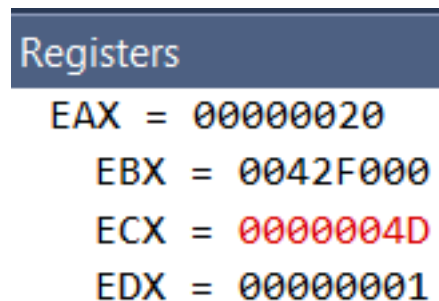
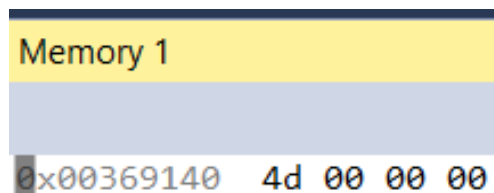


Figure 24d: Contents of register ECX

This register contains the value of static variable h . The instruction at address 0x00361738 will perform an addition on the value stored in this register with the value stored at memory address 0x00369168 (i.e. $A[8]$). This is the equivalent of $A[8] + h$ in C.

Figure 24e: Contents of register ECX after $A[8] + h$

Lastly, this result is stored at memory address location 0x00369140 (i.e. the location of variable g) and the program terminates.

Figure 24f: Memory window showing the location and value of g

2-5_2.c (Arrays)

```

void main() {
    static int h = 20;
    static int A[300];
    A[300] = 13;
    A[300] = h + A[300];
}

static int h = 20;
static int A[300];
A[300] = 13;
00521718  mov     eax,4
0052171D  imul    ecx,eax,12Ch
00521723  mov     dword ptr A (0529140h)[ecx],0Dh
    A[300] = h + A[300];
0052172D  mov     eax,4
00521732  imul    ecx,eax,12Ch
00521738  mov     edx,dword ptr [h (0529000h)]
0052173E  add     edx,dword ptr A (0529140h)[ecx]
00521744  mov     eax,4    ≤ 1ms elapsed
00521749  imul    ecx,eax,12Ch
0052174F  mov     dword ptr A (0529140h)[ecx],edx
}

```

Figure 25a: Disassembly Window of 2-5_2.c

The first three instructions at addresses 0x00B81718 to 0x00B81723 will compute the offset from the base address of the address to find `A[300]`. First, the value 4 will be moved into register `EAX`. This value is then multiplied by 0x12C (300) where the result is stored in register `ECX`.

Registers

```

EAX = 00000004
EBX = 00E6C000
ECX = 000004B0

```

Figure 25b: Contents of register `ECX`

This value (1200) will be the offset to find `A[300]`. The assembly code tells us that the base address of `A` is 0x0529140. Using this base address, we add the value stored in the `ECX` register to get 0x05295ED. This the memory location of `A[300]`. The value 0x0D or 13 in decimal, is then stored in this memory address.

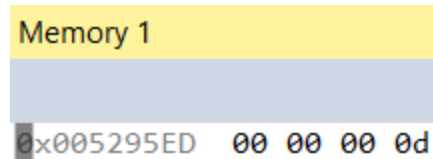


Figure 25c: Memory window showing the location and initial value of $A[300]$

The next instructions at addresses 0x00B8172D and 0x00B81732 repeat the previous process for locating $A[300]$. The next instruction at address 0x00B81738 will move the value stored at memory address 0x00b89000 (i.e. the location of variable h) onto register EDX .

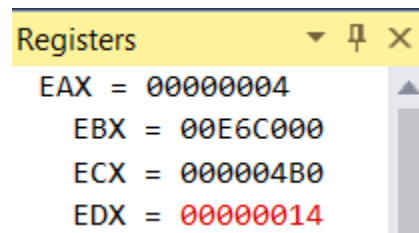


Figure 25d: Contents of register EDX

The instruction at address 0x00B8173E will add the value stored at memory address 0x005295ED to this register and stored back into register EDX . This is the equivalent of in $A[300] + h$ in C.

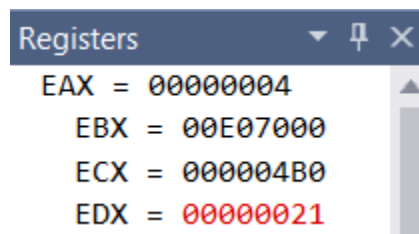


Figure 25e: Updated contents of register EDX after $A[300] + h$

The last three instructions at address 0x00B81744 to 0x00B8174F will compute the address location of $A[300]$ so that the result of the previous addition can be stored at that location.

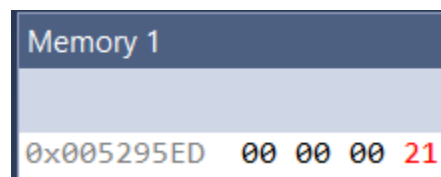


Figure 25f: Memory window showing the location and final value of $A[300]$

The value stored at the memory address of $A[300]$ is now 0x21 or 33 in decimal which is the expected result. The program terminates.

2-6_1.c (Bitwise Operations)

```

void main() {
    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;
}

```

```

static int s0 = 9;
static int t1 = 0x3c00;
static int t2 = 0xdc0;
static int t3 = 0;
t3 = s0 << 4;
00701718  mov     eax,dword ptr [s0 (07090004h)]
0070171D  shl     eax,4
00701720  mov     dword ptr [t3 (0709150h)],eax
static int t0 = 0;
t0 = t1 & t2;
00701725  mov     eax,dword ptr [t1 (07090004h)]
0070172A  and     eax,dword ptr [t2 (07090008h)]
00701730  mov     dword ptr [t0 (0709154h)],eax
t0 = t1 | t2;
00701735  mov     eax,dword ptr [t1 (07090004h)]
0070173A  or      eax,dword ptr [t2 (07090008h)]
00701740  mov     dword ptr [t0 (0709154h)],eax
t0 = ~t1;
00701745  mov     eax,dword ptr [t1 (07090004h)]
0070174A  not     eax
0070174C  mov     dword ptr [t0 (0709154h)],eax
}

```

Figure 26a: Disassembly window of 2-6_1.c

The first instruction at address 0x00701718 will copy the value stored at memory location 0x0079000 (i.e. the location of variable `s0`) onto register EAX.

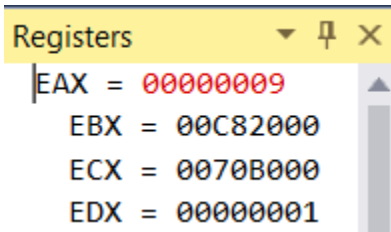


Figure 26b: Contents of register EAX

Register EAX now contains the value of `s0` (i.e. 9). The instruction at address 0x0070171D will perform a left shift by 4 bytes on the value stored in register EAX.

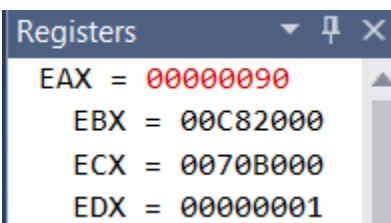


Figure 26c: Updated Contents of register EAX after left shift

The instruction at address 0x0070171D will store the current contents of register EAX to memory location 0x00709150 (i.e. the location for variable `t3`)

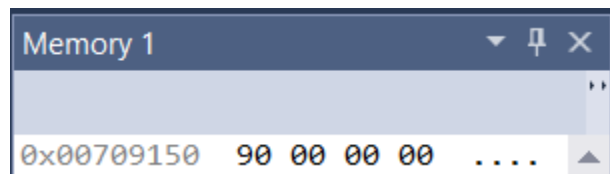


Figure 26d: Memory location and value of variable `t3`

The instructions at addresses 0x00701725 to 0x00701730 will perform the C equivalent operation `t0 = t1 & t2`. First, the value stored at the memory address of `t0` will be copied to register EAX. Next, a bitwise AND operation will be performed on the value currently stored in EAX and the memory value stored in `t2`. The result will then stored back into register EAX.

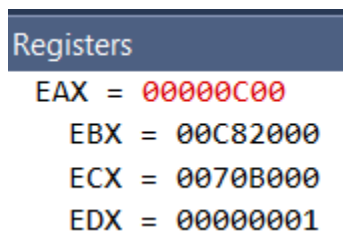


Figure 26e: Updated Contents of register EAX after bitwise AND

Lastly, the result will be copied to memory address 0x00709154 (i.e. the location of variable `t0`).

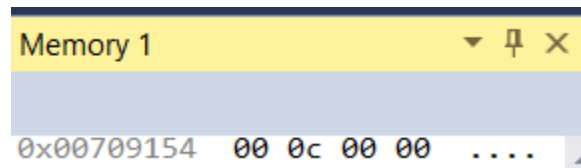


Figure 26f: Memory location and value of variable `t0`

The next instructions at addresses 0x00709135 to 0x00709140 are similar to the previous instructions except that a bitwise OR is performed instead. The C equivalent of these instructions are `t0 = t1 | t2`. The result of the bitwise OR is stored onto register and `EAX` then in the memory address 0x00709154 (i.e. the location of variable `t0`).

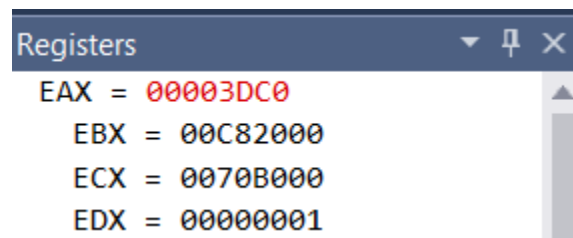


Figure 26g: Updated Contents of register `EAX` after bitwise OR

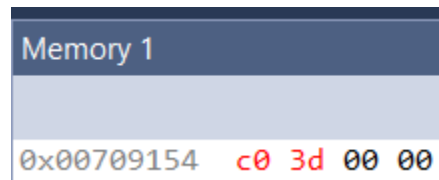


Figure 26h: Memory location and updated value of variable `t0`

The next instructions at addresses 0x00709145 to 0x0070914C are similar to the previous instructions except that a bitwise OR is performed instead. The C equivalent of these instructions are `t0 = ~t1`. The result of the bitwise OR is stored onto register and `EAX` then in the memory address 0x00709154 (i.e. the location of variable `t0`).

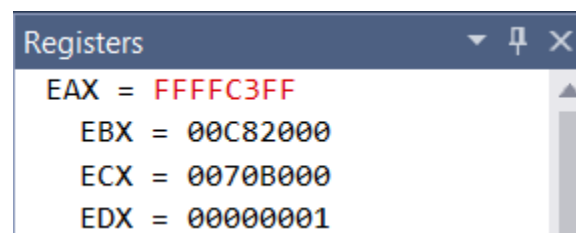


Figure 26i: Updated Contents of register `EAX` after bitwise NOR

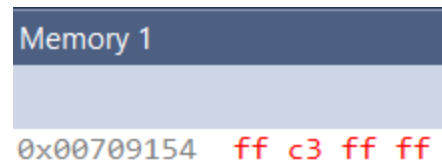


Figure 26j: Memory location and updated value of variable t_0

All three bitwise operations have been performed and so the program terminates.

If_Else.c

```

int main() {
    int s0 = 5;
    int s1 = 10;
    int t0;

    t0 = s0;
    s0 = s1;
    s1 = t0;
}

static int a = 0;
static int b = 100;
static int c = 80;
static int d = 60;

    if (a == d) {
007A1718  mov     eax,dword ptr [a (07A9150h)]
007A171D  cmp     eax,dword ptr [d (07A9008h)]
007A1723  jne     main+47h (07A1737h)
        a = b + c;
007A1725  mov     eax,dword ptr [b (07A9000h)]
007A172A  add     eax,dword ptr [c (07A9004h)]
007A1730  mov     dword ptr [a (07A9150h)],eax
    }
    else {
007A1735  jmp     main+57h (07A1747h)
        a = b - c;
007A1737  mov     eax,dword ptr [b (07A9000h)]
007A173C  sub     eax,dword ptr [c (07A9004h)]
007A1742  mov     dword ptr [a (07A9150h)],eax
    }
}

```

Figure 27a: Disassembly window of If_Else.c

At the start of the program, static variable `a` will be stored at memory address `0x007A9150` with initial value 0.

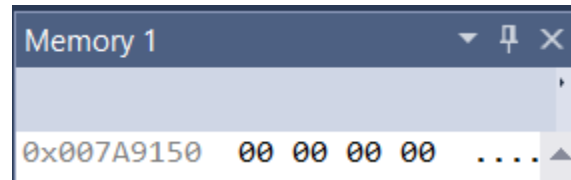


Figure 27b: Memory window showing the location and initial value of variable `a`

The instruction at address `0x007A1718` will copy the value stored at the memory address `0x007A9150` (i.e. static variable `a`) onto register `EAX`.

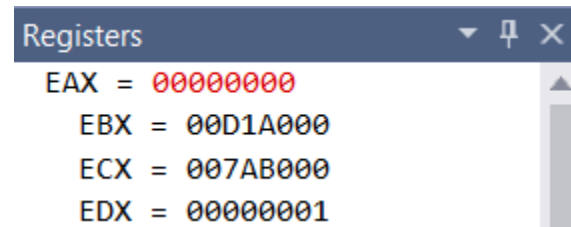


Figure 27c: Register window showing the contents of register `EAX`

The instruction at address `0x007A171D` will compare the value stored in register `EAX` (i.e. static variable `a`) with the value stored at memory address `0x007A9008` (i.e. static variable `d = 60`). Since these two values are not equal, a jump occurs from this address to address `0x007A1737`. At this address, the value stored at the memory address `0x007A9000` (i.e. static variable `b`) is copied onto register `EAX`.

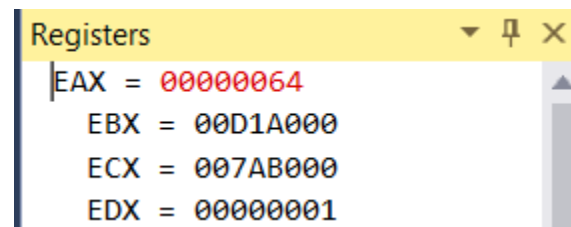


Figure 27d: Register window showing the updated contents of register `EAX`

The instruction at address `0x007A173C` will perform a subtraction on the contents of register `EAX` and the value stored at memory address `0x007A9004` (i.e. static variable `c`) and store the result back into register `EAX`. This is the C equivalent of `b-c`.

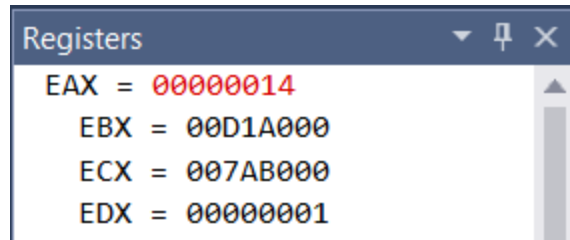


Figure 27e: Register window showing the updated contents of register EAX

The value stored at register EAX is now 20 in decimal. This is the expected result in the values of static variables b and c are 100 and 80 respectively. Thus, $100 - 80 = 20$. This result is the value of static variable a. The next instruction at address 0x007A1742 will copy result of the subtraction into the memory address 0x007A9150 (i.e. the memory location of a).

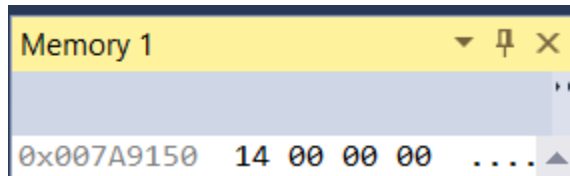


Figure 27f: Memory window showing the location and final value of a

WhileLoop.c

```

void main() {
    static int a = 0;
    static int b = 10;

    while (a <= b) {
        a++;
    }

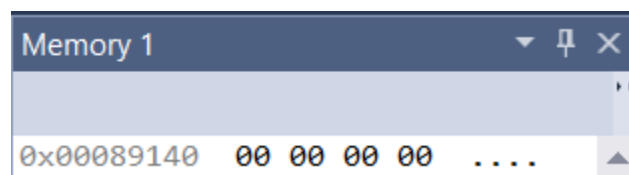
    static int a = 0;
    static int b = 10;

    while (a <= b) {
00081718  mov     eax,dword ptr [a (089140h)]
0008171D  cmp     eax,dword ptr [b (089000h)]
00081723  jg      main+44h (081734h)
        a++;
00081725  mov     eax,dword ptr [a (089140h)]
0008172A  add     eax,1
0008172D  mov     dword ptr [a (089140h)],eax
    }
00081732  jmp     main+28h (081718h)
}

```

Figure 28a: Disassembly window of WhileLoop.c

At the start of the program, static variable `a` will be stored at memory address `0x00089140` with initial value `0`.

Figure 28b: Memory window showing the location and initial value of variable `a`

The instruction at address `0x00081718` will copy the value stored at the memory address `0x00089140` (i.e. static variable `a`) onto register `EAX`.

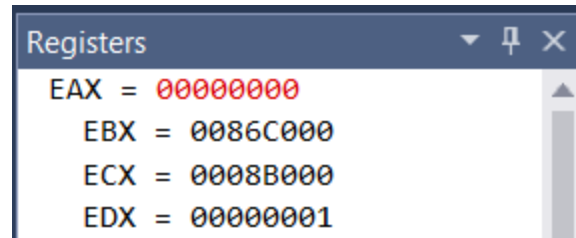


Figure 28c: Register window showing the contents of register EAX

The next instruction at address 0x0008171D will compare the value stored in register EAX (i.e. static variable a) with the value stored at memory address 0x00089000 (i.e. static variable b = 10). Since $a < b$ we jump into the while loop and the instructions at addresses 0x00081725 and 0x0008172A add 1 to the value stored in register EAX. This is the C equivalent of `a++`.

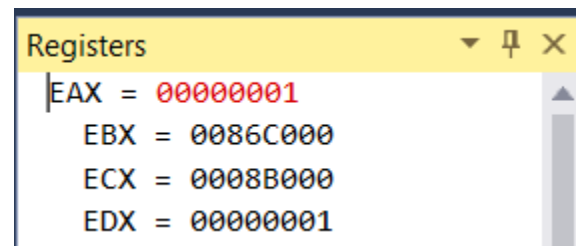


Figure 28d: Register window showing the updated contents of register EAX after first while loop iteration

The next instruction at address 0x0008172D will copy the updated contents of register EAX into the memory address 0x00089140 (i.e. the memory location of a).

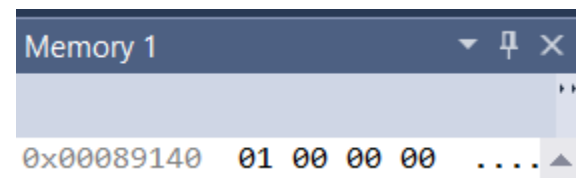


Figure 28e: Memory window showing the location and value of variable a after first while loop iteration

This process repeats until the condition for the while loop becomes false. In the final iteration, the value stored in the register is 11 in decimal.

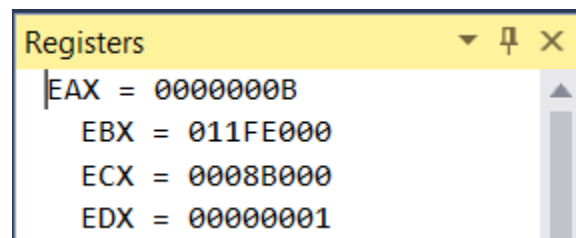


Figure 28f: Register window showing the updated contents of register EAX after final while loop iteration

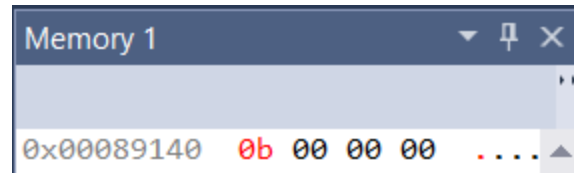


Figure 28g: Memory window showing the location and value of variable `a` after final while loop iteration

Local.c

```

int main() {
    int s0 = 5;
    int s1 = 10;
    int t0;

    t0 = s0;
    s0 = s1;
    s1 = t0;
}

```

```

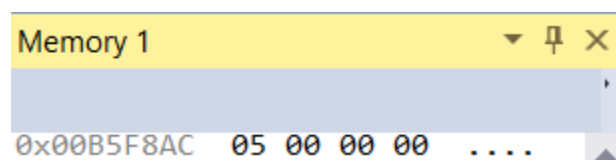
    int s0 = 5;
002B1718 mov          dword ptr [s0],5
    int s1 = 10;
002B171F mov          dword ptr [s1],0Ah
    int t0;

    t0 = s0;
002B1726 mov          eax,dword ptr [s0]
002B1729 mov          dword ptr [t0],eax
    s0 = s1;
002B172C mov          eax,dword ptr [s1]
002B172F mov          dword ptr [s0],eax
    s1 = t0;
002B1732 mov          eax,dword ptr [t0]
002B1735 mov          dword ptr [s1],eax
}

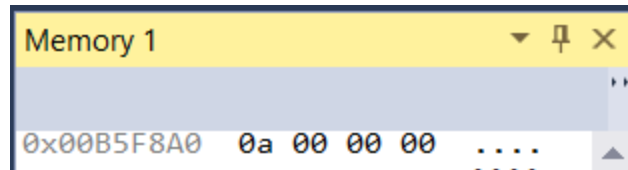
```

Figure 29a: Disassembly window of Local.c

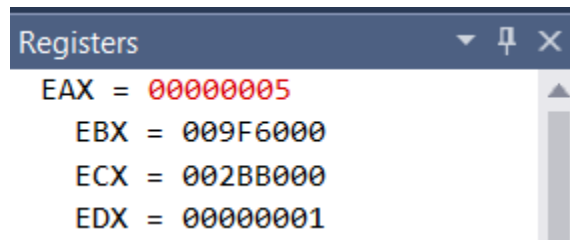
The instruction at address 0x0002B1718 will load the *local* variable `s0` onto the stack with initialized value of 5 in decimal.

Figure 29b: Initializing `s0` on stack

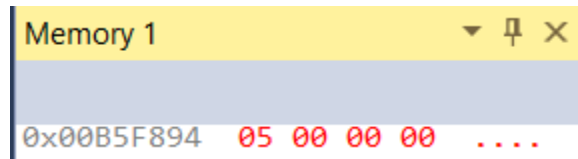
The instruction at address 0x0002171F will load the *local* variable `s1` onto the stack with initialized value of 10 in decimal.

Figure 29c: Initializing `s1` on stack

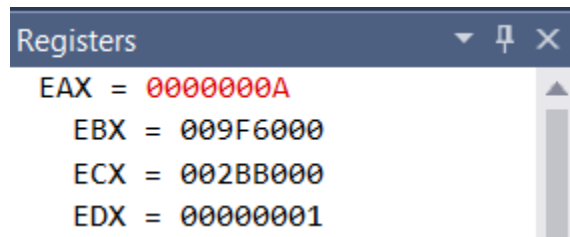
The instruction at address 0x00021726 will load the memory value of `s0` into register `EAX`.

Figure 29d: Register window showing the contents of register `EAX`

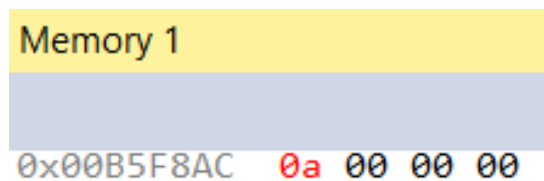
The instruction at address 0x00021726 will store the value of this register into memory location of `t0`. This is the C equivalent of `t0 = s0`.

Figure 29e: Location and value of `t0` in memory

The instruction at address 0x0002172C will load the memory value of `s1` into register `EAX`.

Figure 29f: Register window showing the contents of register `EAX`

The instruction at address 0x0002172F will store the value of this register into memory location of `s0`. This is the C equivalent of `s0 = s1`.

Figure 29g: Location and value of `s0` in memory

The instructions at address 0x00021732 will load the memory value of `t0` into register `EAX`.

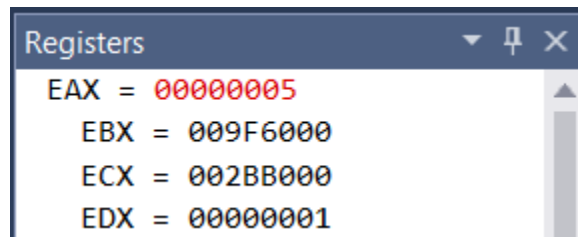


Figure 29h: Register window showing the contents of register `EAX`

The instruction at address 0x00021735 will store the value of this register into memory location of `s1`. This is the C equivalent of `s1 = s0`.

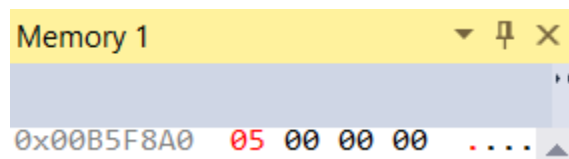


Figure 29i: Location and value of `s1` in memory

The swapping of contents of `s0` and `s1` is now complete.

Natural_Generator.c

```
#include <stdio.h>

int natural_generator() {
    int a = 1;
    static int b = -1;
    b += 1;
    return a + b;
}

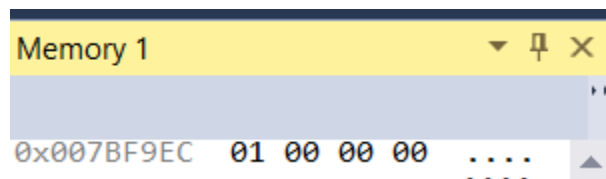
int main() {
    printf("%d\n", natural_generator());
    printf("%d\n", natural_generator());
    printf("%d\n", natural_generator());

    return 0;
}
```

```
    int a = 1;
006B1718  mov         dword ptr [a],1
    static int b = -1;
    b += 1;
006B171F  mov         eax,dword ptr [b (06B9000h)]
006B1724  add         eax,1
006B1727  mov         dword ptr [b (06B9000h)],eax
    return a + b;
006B172C  mov         eax,dword ptr [a]
006B172F  add         eax,dword ptr [b (06B9000h)]
}
```

Figure 30a: Disassembly window of Local.c

The instruction at address 0x006B1718 will load the value of *local* variable *a* onto location 0x007BF9EC in memory.

Figure 30b: Initializing *local* variable *a*

The instruction at address 0x006B171F will copy the value stored at memory address 0x006171F onto register EAX.

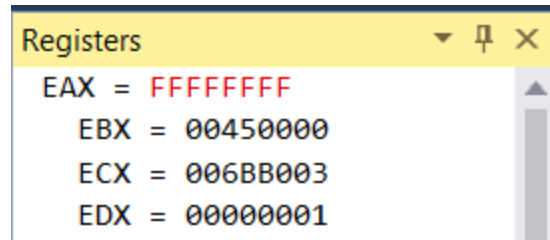


Figure 30c: Register window showing the contents of register EAX

The instruction at address 0x006B1724 will add 1 to the current value stored in register EAX.

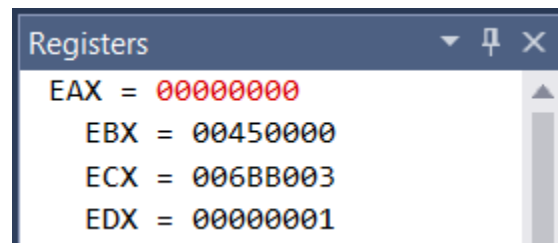


Figure 30d: Register window showing the updated contents of register EAX

The instruction at address 0x006B1727 will store the value of this register into memory location of b.

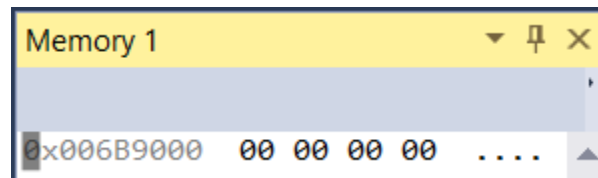


Figure 30e: Memory window showing the location and value of variable b

The instruction at address 0x006B172C will copy the value stored at memory address 0x007BF9EC (i.e. the value of local variable a) onto register EAX. The next instruction at address 0x006B172F will add the value stored at memory location 0x006B9000 (i.e. the value of static variable b) to the current contents of the register. This is the C equivalent of $a+b$.

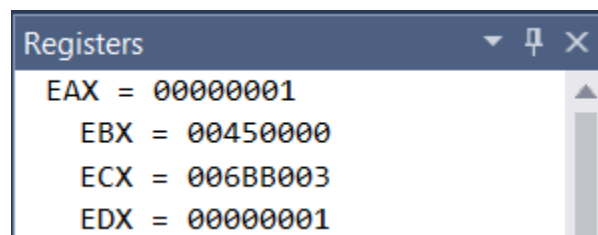


Figure 30f: Register window showing the updated contents of register EAX after $a+b$

It should be noted that the result of the addition is not stored back into memory. The program terminates.

Part III: Intel X86 64-bit on Linux platform using GDB

Topics Covered: Static/Local variables, Arrays, Bitwise Operations, Condition Statements, Loops

Debugging a C program in Linux using GDB

In this final part, the previously tested programs will be analyzed in a Linux environment. Unlike the previous two environments, there is no GUI that we can interact with. Instead, all analysis will take place within the command line using various commands. To begin running a C program using gdb, it must first be compiled in the following way:

```
gcc -g -o0 <filename> -o <name>
```

Then to begin using gdb, we run the command

```
gdb <name>
```

To step through the C code we run `break main` followed by `run`. To run one line of source code, we run `next`. Likewise to run one assembly instruction, we run `stepi`. To view the assembly code we run `disassemble`. The assembly instructions shown are in AT&T syntax formatted as `source, destination`. The Linux environment comprised of Ubuntu 18.04 (Bionic Beaver), gcc version 7.5.0 and gdb version 8.1.

2-2_1.c (Simple Arithmetic)

The image below shows the assembly code of this program after compilation.

```

Dump of assembler code for function main:
0x00005555555545fa <+0>:      push    %rbp
0x00005555555545fb <+1>:      mov     %rsp,%rbp
=> 0x00005555555545fe <+4>:      mov     0x200a0c(%rip),%edx      # 0x555555755010 <b.1795>
0x0000555555554604 <+10>:     mov     0x200a0a(%rip),%eax      # 0x555555755014 <c.1796>
0x000055555555460a <+16>:     add     %edx,%eax
0x000055555555460c <+18>:     mov     %eax,0x200a06(%rip)      # 0x555555755018 <a.1794>
0x0000555555554612 <+24>:     mov     0x200a00(%rip),%edx      # 0x555555755018 <a.1794>
0x0000555555554618 <+30>:     mov     0x2009fe(%rip),%eax      # 0x55555575501c <e.1798>
0x000055555555461e <+36>:     sub     %eax,%edx
0x0000555555554620 <+38>:     mov     %edx,%eax
0x0000555555554622 <+40>:     mov     %eax,0x2009f8(%rip)      # 0x555555755020 <d.1797>
0x0000555555554628 <+46>:     nop
0x0000555555554629 <+47>:     pop     %rbp
0x000055555555462a <+48>:     retq
End of assembler dump.

```

Figure 31a: Assembly code for 2-2_1.c

Initially, the first two instructions (which are executed by the time we start debugging) are considered the function prologue. It should be the case that the values of the Stack Pointer (\$rsp) and Base Pointer (\$rbp) registers are the same. This is the case because the old base pointer is pushed onto the stack for later use. Then, the value of the stack pointer is copied to the base pointer. To verify this, we make use of the `print /x` command and print out the register values in hexadecimal.

```

(gdb) print /x $rbp
$1 = 0x7fffffffdf50
(gdb) print /x $rsp
$2 = 0x7fffffffdf50

```

Figure 31b: Printing the contents of \$rbp and \$rsp

The address of the next instruction is given by the => symbol. Alternatively, we can check for the next instruction by running the `print /x $rip` which prints the contents of the Instruction Pointer register.

```

(gdb) print /x $rip
$3 = 0x5555555545fe

```

Figure 31c: Printing the contents of \$rip

Note that this value matches exactly with the value given by \Rightarrow in figure 31c above. In the next two instructions (boxed in green), the values of variables `b` and `c` are moved to registers `%edx` and `%eax` respectively.

```
(gdb) print /x $edx
$4 = 0x2
```

```
(gdb) print /x $eax
$5 = 0x3
```

Figure 31d: Printing the contents of `$edx` and `$eax`

The next instructions (boxed in orange) perform an addition on the contents of the `%edx` and `%eax` registers and stores the result back into the `%eax` register. Figure 31e shows the updated contents of `%eax`.

```
(gdb) print /x $eax
$6 = 0x5
```

Figure 31e: Printing the updated contents of `$eax`

The value stored in register `%eax` is now 5 in decimal which is the expected result since $2 + 3 = 5$.

```
(gdb) x/8xb 0x555555755010
0x555555755010 <b.1795>:    0x02    0x00    0x00    0x00    0x03    0x00    0x00    0x00
(gdb) x/8xb 0x555555755014
0x555555755014 <c.1796>:    0x03    0x00    0x00    0x00    0x05    0x00    0x00    0x00
(gdb) x/8xb 0x555555755018
0x555555755018 <a.1794>:    0x05    0x00    0x00    0x00    0x05    0x00    0x00    0x00
```

Figure 31f: Memory values of variables `a`, `b`, and `c`

Figure 31f above shows the memory values `a`, `b`, and `c`. The memory addresses are given by the `%rip` register with some offset. The next instructions (boxed in blue) will store the values of variables `d` and `e` to registers `%edx` and `%eax` respectively.

```
(gdb) print /x $edx
$8 = 0x5
```

```
(gdb) print /x $eax
$9 = 0x5
```

Figure 31g: Printing the contents of `$edx` and `$eax`

The next instructions (boxed in white) perform a subtraction on the contents of the `%edx` and `%eax` registers and stores the result back into the `%edx` register. Afterwards, the value of `%edx` is

copied into `%eax`. Figure 31h shows the updated contents of `%eax` and `%edx`. The value stored in register `%eax` is now 0 in decimal which is the expected result since $5 - 5 = 0$.

```
(gdb) print /x $eax (gdb) print /x $edx
$11 = 0x0             $12 = 0x0
```

Figure 31h: Printing the updated contents of `$eax`

Figure 31i shows the memory values `a`, `d`, and `e`.

```
0x555555755018 <a.1794>:      0x05    0x00    0x00    0x00    0x05    0x00    0x00    0x00
(gdb) x/8xb 0x55555575501c
0x55555575501c <e.1798>:      0x05    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) x/8xb 0x555555755020
0x555555755020 <d.1797>:      0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

Figure 31i: Memory values of variables `a`, `d`, and `e`

The next instruction `nop` performs no operation. The following instructions (boxed in yellow) will pop the old base pointer from the stack and store it back into register `$ebp`. The very last instruction will cause us to jump back to our initial stack frame and the program terminates.

2-3_1.c (Arrays)

```

Dump of assembler code for function main:
0x00005555555545fa <+0>:    push    %rbp
0x00005555555545fb <+1>:    mov     %rsp,%rbp
=> 0x00005555555545fe <+4>:    movl    $0x37,0x200a58(%rip)    # 0x555555755060 <A.1796+32>
0x0000555555554608 <+14>:    mov     0x200a52(%rip),%edx    # 0x555555755060 <A.1796+32>
0x000055555555460e <+20>:    mov     0x2009fc(%rip),%eax    # 0x555555755010 <h.1795>
0x0000555555554614 <+26>:    add     %edx,%eax
0x0000555555554616 <+28>:    mov     %eax,0x200bb4(%rip)    # 0x5555557551d0 <g.1794>
0x000055555555461c <+34>:    nop
0x000055555555461d <+35>:    pop     %rbp
0x000055555555461e <+36>:    retq
End of assembler dump.

```

Figure 32a: Assembly code for 2-3_1.c

After the first two instructions, the next instruction (boxed in green) will move the value 0x37 or 55 in decimal to memory location 0x755060 with +32 offset which is the location of `A[8]`. That is, `A[8] = 55`.

```

(gdb) x/8xb 0x555555755060
0x555555755060 <A.1796+32>:    0x37    0x00    0x00    0x00    0x00    0x00    0x00    0x00

```

Figure 32b: Memory location of `A[8]`.

The next two instructions (boxed in orange) will store values `A[8]` and `0x16` (i.e. the value of static variable `h`) in registers `%edx` and `%eax` respectively.

```

(gdb) print /x $edx
$5 = 0x37
(gdb) print /x $eax
$6 = 0x16

```

Figure 32c: Printing the contents of `$edx` and `$eax`

The next instructions (boxed in blue) will perform an addition between the contents of the two registers above and store the result back into register `$eax`.

```

(gdb) print /x $eax
$7 = 0x4d

```

Figure 32d: Printing the updated contents `$eax`

The value now stored in `$eax` is 0x4d or 77 which is expected because the contents of registers `%edx` and `%eax` were 55 and 22 respectively in decimal (i.e. $55 + 22 = 77$). The result is then moved into memory location 0x7551D0 (i.e. the location of variable `g`).

```
(gdb) x/8xb 0x5555557551d0
0x5555557551d0 <g.1794>:    0x4d    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

Figure 32e: Memory location of `A[8]`.

The final three instructions return the stack frame to its initial state. The program terminates.

2-5_2.c (Arrays)

```

Dump of assembler code for function main:
0x00005555555545fa <+0>:    push    %rbp
0x00005555555545fb <+1>:    mov     %rsp,%rbp
=> 0x00005555555545fe <+4>:    movl    $0xd,0x200ee8(%rip)    # 0x5555557554f0 <A.1795+1200>
0x0000555555554608 <+14>:   mov     0x200ee2(%rip),%edx    # 0x5555557554f0 <A.1795+1200>
0x000055555555460e <+20>:   mov     0x2009fc(%rip),%eax    # 0x555555755010 <h.1794>
0x0000555555554614 <+26>:   add     %edx,%eax
0x0000555555554616 <+28>:   mov     %eax,0x200ed4(%rip)    # 0x5555557554f0 <A.1795+1200>
0x000055555555461c <+34>:   nop
0x000055555555461d <+35>:   pop     %rbp
0x000055555555461e <+36>:   retq
End of assembler dump.

```

Figure 33a: Assembly code for 2-5_2.c

After the first two instructions, the next instruction (boxed in green) will move the value 0xD or 13 in decimal to memory location 0x7554F0 with +1200 offset which is the location of $A[300]$. That is, $A[300] = 13$.

```

(gdb) x/8xb 0x5555557554f0
0x5555557554f0 <A.1795+1200>: 0x0d 0x00 0x00 0x00 0x00 0x00 0x00 0x00

```

Figure 33b: Memory location and initial value of $A[300]$.

The next instructions (boxed in orange) will move the memory values at addresses 0x7554F0 and 0x755010 (i.e. the locations of $A[300]$ and h) to registers $\$edx$ and $\$eax$ respectively.

```

(gdb) print /x $edx
$1 = 0xd
(gdb) print /x $eax
$2 = 0x14

```

Figure 33c: Printing the contents of $\$edx$ and $\$eax$

These two values are added where the result is stored back into register $\$eax$. This is the C equivalent of $A[300] = A[300] + h$.

```

(gdb) print /x $eax
$3 = 0x21

```

Figure 33d: Printing the updated contents of $\$eax$

The result is 0x21 or 33 in decimal which is the expected result. The final instruction (boxed in blue) will store the result into the memory address 0x7554F0 (i.e. the location of $A[300]$).

```

0x5555557554f0 <A.1795+1200>: 0x21 0x00 0x00 0x00 0x00 0x00 0x00 0x00
(gdb)

```

Figure : Memory location and final value of $A[300]$.

2-6_1.c (Bitwise Operations)

The image below shows the assembly code of this program after compilation.

```
Dump of assembler code for function main:
0x00005555555545fa <+0>:    push    %rbp
0x00005555555545fb <+1>:    mov     %rsp,%rbp
=> 0x00005555555545fe <+4>:    mov     0x200a0c(%rip),%eax    # 0x555555755010 <s0.1794>
0x0000555555554604 <+10>:   shl     $0x4,%eax
0x0000555555554607 <+13>:   mov     %eax,0x200a13(%rip)    # 0x555555755020 <t3.1797>
0x000055555555460d <+19>:   mov     0x200a01(%rip),%edx    # 0x555555755014 <t1.1795>
0x0000555555554613 <+25>:   mov     0x2009ff(%rip),%eax    # 0x555555755018 <t2.1796>
0x0000555555554619 <+31>:   and     %edx,%eax
0x000055555555461b <+33>:   mov     %eax,0x200a03(%rip)    # 0x555555755024 <t0.1798>
0x0000555555554621 <+39>:   mov     0x2009ed(%rip),%edx    # 0x555555755014 <t1.1795>
0x0000555555554627 <+45>:   mov     0x2009eb(%rip),%eax    # 0x555555755018 <t2.1796>
0x000055555555462d <+51>:   or      %edx,%eax
0x000055555555462f <+53>:   mov     %eax,0x2009ef(%rip)    # 0x555555755024 <t0.1798>
0x0000555555554635 <+59>:   mov     0x2009d9(%rip),%eax    # 0x555555755014 <t1.1795>
0x000055555555463b <+65>:   not     %eax
0x000055555555463d <+67>:   mov     %eax,0x2009e1(%rip)    # 0x555555755024 <t0.1798>
0x0000555555554643 <+73>:   nop
0x0000555555554644 <+74>:   pop     %rbp
0x0000555555554645 <+75>:   retq
End of assembler dump.
```

Figure 34a: Assembly code for 2-3_1.c

Upon executing the initial two instructions, the next instructions (boxed in green) will begin by moving a value from memory onto register `$eax`.

```
(gdb) print /x $eax
$1 = 0x9
```

Figure 34b: Printing the contents of `$eax`

The following instruction will perform a left shift by 4 bits on the contents of register `$eax`.

```
(gdb) print /x $eax
$2 = 0x90
```

Figure 34c: Printing the updated contents of `$eax` after left shift

This value will be then loaded into memory address `0x755020`. The figure below shows result in memory before and after the left shift. The memory addresses `0x755010` and `0x755020` are the locations for `s0` and `t3` respectively.

```
(gdb) x/8xb 0x555555755010
0x555555755010 <s0.1794>:    0x09    0x00    0x00    0x00    0x00    0x3c    0x00    0x00
(gdb) x/8xb 0x555555755020
0x555555755020 <t3.1797>:    0x90     0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

Figure 34d: The locations and values of `s0` and `t3` in memory

The next instructions (boxed in orange) perform the C equivalent of $t0 = t1 \& t2$. First, the values at memory addresses 0x755014 and 0x755018 are copied to registers `$edx` and `$eax` respectively.

```
(gdb) print /x $edx
$3 = 0x3c00
(gdb) print /x $eax
$4 = 0xdc0
```

Figure 34e: Printing the contents of `$edx` and `$eax` containing variables `t1` and `t2`

Next, the bitwise AND operation is performed where the result is stored back into register `$eax`.

```
(gdb) print /x $eax
$5 = 0xc00
```

Figure 34f: Printing the updated contents of `$eax` after bitwise AND

Lastly, the result is copied into memory address 0x755024.

```
(gdb) x/8xb 0x555555755024
0x555555755024 <t0.1798>: 0x00 0x0c 0x00 0x00 0x00 0x00 0x00 0x00
```

Figure 34g: The location and value of `t0` in memory

The next set of instructions (boxed in blue) perform the C equivalent of $t0 = t1 | t2$. First, the values at memory addresses 0x755014 and 0x755018 are copied to registers `$edx` and `$eax` respectively. Next, the bitwise OR operation is performed where the result is stored back into register `$eax`.

```
(gdb) print /x $eax
$8 = 0x3dc0
```

Figure 34g: Printing the updated contents of `$eax` after bitwise OR

Lastly, the result is copied into memory address 0x755024.

```
(gdb) x/8xb 0x555555755024
0x555555755024 <t0.1798>: 0xc0 0x3d 0x00 0x00 0x00 0x00 0x00 0x00
```

Figure 34h: The location and updated value of `t0` in memory

The last set of instructions (boxed in white) perform the C equivalent of $t0 = \sim t1$. First, the values at memory addresses 0x755014 and 0x755018 are copied to registers `$edx` and `$eax` respectively. Next, the bitwise NOT operation is performed where the result is stored back into register `$eax`.

```
(gdb) print /x $eax
$9 = 0xfffffc3ff
```

Figure 34i: Printing the updated contents of `$eax` after bitwise NOT

Lastly, the result is copied into memory address `0x755024`.

```
(gdb) x/8xb 0x555555755024
0x555555755024 <t0.1798>:  0xff  0xc3  0xff  0xff  0x00  0x00  0x00  0x00
```

Figure 34j: The location and updated value of `t0` in memory

All three bitwise operations have been performed and the so the program terminate

If_Else.c

The image below shows the assembly code of this program after compilation.

```

Dump of assembler code for function main:
0x00005555555545fa <+0>:      push    %rbp
0x00005555555545fb <+1>:      mov     %rsp,%rbp
=> 0x00005555555545fe <+4>:      mov     0x200a1c(%rip),%edx      # 0x555555755020 <a.1794>
0x0000555555554604 <+10>:     mov     0x200a06(%rip),%eax      # 0x555555755010 <d.1797>
0x000055555555460a <+16>:     cmp     %eax,%edx
0x000055555555460c <+18>:     jne     0x555555554624 <main+42>
0x000055555555460e <+20>:     mov     0x200a00(%rip),%edx      # 0x555555755014 <b.1795>
0x0000555555554614 <+26>:     mov     0x2009fe(%rip),%eax      # 0x555555755018 <c.1796>
0x000055555555461a <+32>:     add     %edx,%eax
0x000055555555461c <+34>:     mov     %eax,0x2009fe(%rip)      # 0x555555755020 <a.1794>
0x0000555555554622 <+40>:     jmp     0x55555555463a <main+64>
0x0000555555554624 <+42>:     mov     0x2009ea(%rip),%edx      # 0x555555755014 <b.1795>
0x000055555555462a <+48>:     mov     0x2009e8(%rip),%eax      # 0x555555755018 <c.1796>
0x0000555555554630 <+54>:     sub     %eax,%edx
0x0000555555554632 <+56>:     mov     %edx,%eax
0x0000555555554634 <+58>:     mov     %eax,0x2009e6(%rip)      # 0x555555755020 <a.1794>
0x000055555555463a <+64>:     nop
0x000055555555463b <+65>:     pop     %rbp
0x000055555555463c <+66>:     retq
End of assembler dump.

```

Figure 35a: Assembly code for If_Else.c

After the initial two instructions, the next instructions (boxed in green) will move the values of static variables `a` and `d` from memory into registers `$edx` and `$eax` respectively.

```

(gdb) print /x $edx
$1 = 0x0
(gdb) print /x $eax
$2 = 0x3c

```

Figure 35b: Printing the contents of `$edx` and `$eax` containing variables `a` and `d`

The next series of instructions (boxed in orange) The contents of these two registers will be compared via `cmp`. We already know that the register contents are not equal and so we expect a jump to occur from the instruction with offset +18 to the instruction with offset +42. This jump indeed occurs skipping the instructions boxed in lilac. The next set of instructions (boxed in blue) will move static variables `b` and `c` from memory into registers `$edx` and `$eax` respectively.

```

(gdb) print /x $edx
$3 = 0x64
(gdb) print /x $eax
$4 = 0x50

```

Figure 35c: Printing the contents of `$edx` and `$eax` containing `b` and `c`

The next instructions (boxed in white) performs a subtraction on the contents of the two registers and stores the result back into register `$edx`. The value of `$edx` is then copied into register `$edx`.

```
(gdb) print /x $edx
$7 = 0x14
(gdb) print /x $eax
$8 = 0x14
```

Figure 35d: Printing the contents of `$edx` and `$eax` each containing the result of `b-c`

Both registers now contain the result the subtraction of `$edx-$eax` (i.e. `b-c`). The value is `0x14` or 20 in decimal which is the expected result. The next instruction (boxed in yellow) will move the contents of register `$eax` to memory location `0x555555755020`.

```
(gdb) x/8xb 0x555555755020
0x555555755020 <a.1794>: 0x14 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

Figure 35e: The locations `$eax` in memory

The final set of instructions return the stack frame to its initial state and the program terminates.

WhileLoop.c

```

Dump of assembler code for function main:
0x00005555555545fa <+0>:    push    %rbp
0x00005555555545fb <+1>:    mov     %rsp,%rbp
=> 0x00005555555545fe <+4>:    jmp     0x55555555460f <main+21>
0x0000555555554600 <+6>:    mov     0x200a12(%rip),%eax    # 0x555555755018 <a.1794>
0x0000555555554606 <+12>:   add     $0x1,%eax
0x0000555555554609 <+15>:   mov     %eax,0x200a09(%rip)    # 0x555555755018 <a.1794>
0x000055555555460f <+21>:   mov     0x200a03(%rip),%edx    # 0x555555755018 <a.1794>
0x0000555555554615 <+27>:   mov     0x2009f5(%rip),%eax    # 0x555555755010 <b.1795>
0x000055555555461b <+33>:   cmp     %eax,%edx
0x000055555555461d <+35>:   jle     0x555555554600 <main+6>
0x000055555555461f <+37>:   nop
0x0000555555554620 <+38>:   pop     %rbp
0x0000555555554621 <+39>:   retq
End of assembler dump.

```

Figure 36a: Assembly code for WhileLoop.c

After the initial two instructions, the next instructions (boxed in green) will first jump to from the instruction with offset <+4> to the instruction with offset <+21>. The next instructions will move the values of static variables `a` and `b` from memory into registers `$edx` and `$eax` respectively.

```

(gdb) print /x $edx
$1 = 0x0
(gdb) print /x $eax
$2 = 0xa

```

Figure 36b: Printing the contents of `$edx` and `$eax` containing variables `a` and `b`

The next instruction (boxed in orange) will compare the contents of registers `$edx` and `$eax`. The following instructions (boxed in blue) will execute the while loop for the first time by first moving the contents of register `$eax` from memory, adding 1 to its contents, and moving it back into memory.

```

(gdb) print /x $eax
$3 = 0x1

```

Figure 36c: Printing the contents of `$eax` after first iteration of while loop

```
(gdb) x/8xb 0x555555755018
0x555555755018 <a.1794>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

(gdb) x/8xb 0x555555755018
0x555555755018 <a.1794>: 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

Figure 36d: Location of `$eax` before (top) and after (bottom) first iteration of while loop

The first iteration of the while loop has been executed. The next set of instructions are those boxed in blue. This process repeats until the condition of the while loop is no longer true.

```
(gdb) print /x $eax
$7 = 0xb
```

Figure 36e: Printing the contents of `$eax` after final iteration of while loop

```
(gdb) x/8xb 0x555555755018
0x555555755018 <a.1794>: 0x0b 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

Figure 36f: Location of `$eax` after final iteration of while loop

The final set of instructions return the stack frame to its initial state and the program terminates.

Local.c

```
(gdb) disassemble
Dump of assembler code for function main:
0x00005555555545fa <+0>:    push    %rbp
0x00005555555545fb <+1>:    mov     %rsp,%rbp
=> 0x00005555555545fe <+4>:    movl    $0x5,-0xc(%rbp)
0x0000555555554605 <+11>:   movl    $0xa,-0x8(%rbp)
0x000055555555460c <+18>:   mov     -0xc(%rbp),%eax
0x000055555555460f <+21>:   mov     %eax,-0x4(%rbp)
0x0000555555554612 <+24>:   mov     -0x8(%rbp),%eax
0x0000555555554615 <+27>:   mov     %eax,-0xc(%rbp)
0x0000555555554618 <+30>:   mov     -0x4(%rbp),%eax
0x000055555555461b <+33>:   mov     %eax,-0x8(%rbp)
0x000055555555461e <+36>:   mov     $0x0,%eax
0x0000555555554623 <+41>:   pop     %rbp
0x0000555555554624 <+42>:   retq
End of assembler dump.
```

Figure 37a: Assembly code for Local.c

After the initial two instructions, the next instructions (boxed in green) store the *local* variables `s0` and `s1` onto the stack. Here, the stack grows downwards, so the expression `-0xc(%rbp)` is equivalent to `%rbp - 0xc`. That is to say, local variables `s0` and `s1` are locations `%rbp-0xc` and `%rbp-0x8` on the stack respectively with initial values 5 and 10 respectively.

```
(gdb) x $rbp - 0xc
0x7fffffffdf54: 0x00000005
(gdb) x &s0
0x7fffffffdf54: 0x00000005
```

```
(gdb) x $rbp - 0x8
0x7fffffffdf58: 0x0000000a
(gdb) x &s1
0x7fffffffdf58: 0x0000000a
```

Figure 37b: Locations of local variables `s0` and `s1` on the stack

The next instructions (boxed in orange) begin the swapping process by loading the contents stored at `%rbp-0xC` (i.e. value of local variable `s0`) onto register `$eax`.

```
(gdb) print /x $eax
$1 = 0x5
```

Figure 37c: Printing the contents of `$eax`

This register value is then copied to location `%rbp-0x4` on the stack. This is the location of *local* variable `t0`.

```
0x7fffffffdf5c: 0x00000005
(gdb) x &t0
0x7fffffffdf5c: 0x00000005
```

Figure 37d: Locations of local variable `t0` on the stack

The next set of instructions (boxed in blue) begin by loading the content stored at `%rbp-0x8` (i.e. value of local variable `s0`) onto register `$eax`. This value is then copied to location `%rbp-0xC` on the stack. At this point of the program, `s0 = 10`.

```
(gdb) print /x $eax
$2 = 0xa

(gdb) x $rbp - 0xc
0x7fffffffdf54: 0x0000000a
(gdb) x &s0
0x7fffffffdf54: 0x0000000a
```

Figure 37e: Printing the contents of `$eax` and the updated contents of local variable `s0`

A similar process happens for local variable `s1`. At this point of the program, `s1 = 5`.

```
(gdb) print /x $eax
$3 = 0x5

(gdb) x $rbp - 0x8
0x7fffffffdf58: 0x00000005
(gdb) x &s1
0x7fffffffdf58: 0x00000005
```

Figure 37f: Printing the contents of `$eax` and the updated contents of local variable `s1`

The swapping process is completed. The next instruction simply moves 0 into register `$eax` and the last two instructions provide cleanup.

Natural_Generator.c

```

Dump of assembler code for function natural_generator:
0x000055555555464a <+0>:    push    %rbp
0x000055555555464b <+1>:    mov     %rsp,%rbp
=> 0x000055555555464e <+4>:    movl    $0x1,-0x4(%rbp)
0x0000555555554655 <+11>:   mov     0x2009b5(%rip),%eax    # 0x555555755010 <b.2250>
0x000055555555465b <+17>:   add     $0x1,%eax
0x000055555555465e <+20>:   mov     %eax,0x2009ac(%rip)    # 0x555555755010 <b.2250>
0x0000555555554664 <+26>:   mov     0x2009a6(%rip),%edx    # 0x555555755010 <b.2250>
0x000055555555466a <+32>:   mov     -0x4(%rbp),%eax
0x000055555555466d <+35>:   add     %edx,%eax
0x000055555555466f <+37>:   pop     %rbp
0x0000555555554670 <+38>:   retq
End of assembler dump.

```

Figure 38a: Assembly code for Natural_Generator.c

After the two initial instructions, the instruction boxed in green assigns the value of *local* variable *a*.

```

(gdb) x $rbp - 0x4
0x7fffffffdf4c: 0x00000001
(gdb) x &a
0x7fffffffdf4c: 0x00000001

```

Figure 38b: Memory location and values for local variable *a*

The next instruction (boxed in orange) stores the value of *static* variable *b* into register *\$eax*.

```

(gdb) print /x $eax
$1 = 0xffffffff

```

Figure 38c: Printing the contents of *\$eax*

Note that the value stored in this register is 0xffffffff which is -1 in signed decimal. The value 1 is then added to this register.

```

(gdb) print /x $eax
$2 = 0x0

```

Figure 38d: Printing the updated contents of *\$eax*

The final contents are stored back into memory at address 0x555555755010.

```
(gdb) x/8xb 0x555555755010
0x555555755010 <b.2250>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

Figure 38e: Location of `$eax` (i.e. static variable `b`) in memory

The next instructions (boxed in blue) begins by first moving the contents of static variable `b` onto register `$edx`. Next, local variable `a` is loaded onto register `$eax`.

```
(gdb) print /x $edx
$3 = 0x0

(gdb) print /x $eax
$4 = 0x1
```

Figure 38f: Printing the contents of `$edx` and `$eax`

An addition between the contents of the two registers follows where the result is stored back into register `$eax`.

```
(gdb) print /x $eax
$5 = 0x1
```

Figure 38g: Printing the updated contents of `$eax` after `a + b`

The result is stored is 1 which is expected since adding 0 to a number does not change that number. This result is *NOT* stored back into memory. The next instructions clean up the stack and returns.