|  |
| --- |
|  |
| **Recursive Procedure Calls** |
| Brandon Chin |
|  |
| **CSC342 - Instructor: Prof. Izidor Gertner** |
| **3/30/2015** |

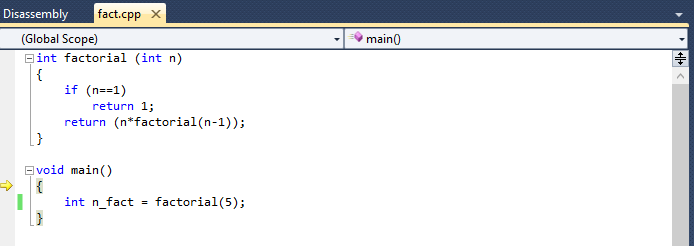
|  |
| --- |
|  |

**Objective:**

Recursion refers to the occurrence of allowing a function to call itself. Meaning the solution to a larger procedure depends on the smaller cases of the same procedure. Inside memory, a new stack frame must be created for each instance of the recursive procedure during execution. This is what we will primarily analyze though debugging and disassembling over three major operating systems; Windows, Linux, and MIPS/MARS.

**Windows OS**

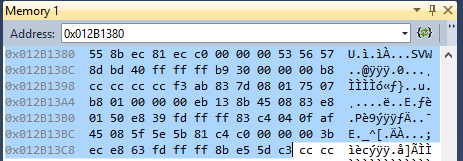
For Windows, we will be using the MS Debugger from Microsoft Visual C++ 2010 Express on a 64-bit Intel Core i5 CPU. Let's debug and analyze the following code segment written in C++:



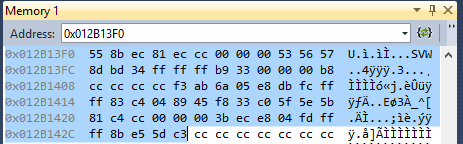
*(C++ Code)*

We have two functions -- the factorial function, which utilizes recursion to calculate the factorial of a value n, and the main function, which calls the factorial function on the value 5. The yellow arrow indicates the location where we will be entering the program (*i.e.* the main function).

The instruction code for our program is compiled and stored into memory.



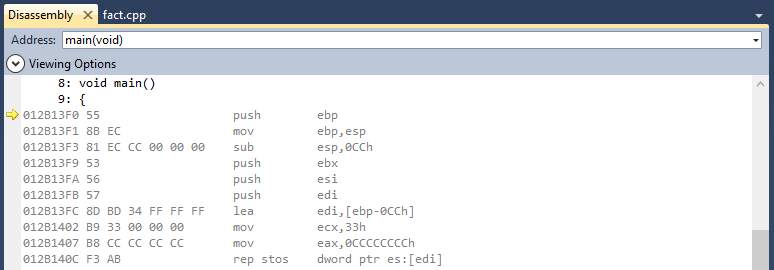
*(Instruction Code - Factorial Function)*



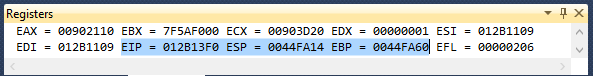
*(Instruction Code - Main Function)*

**Instance #1: Before First Instruction**

Now let's begin debugging. The disassembly of the main function at the instance before any instruction is executed is shown below:

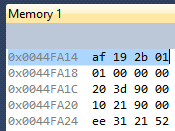
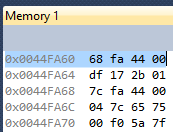


*(Disassembly #1 - main())*



*(Registers #1)*

Let's bring our attention to three specific registers which are highlighted above:

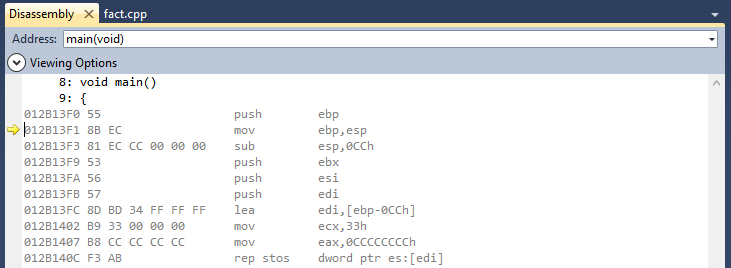
* The EIP stores the address of the instruction which will be executed next. This is also indicated by the yellow arrow displayed next to the same address in the disassembly window.
* The ESP typically stores the address at the top of the stack. In this case, it contains the address which stores the return address that the main procedure must return to after execution is completed.
* The EBP typically stores the address at the base of the stack. In this case, it contains the address which stores the base pointer of the previous calling procedure.  

*(Memory #1 - ESP) (Memory #1 - EBP)*

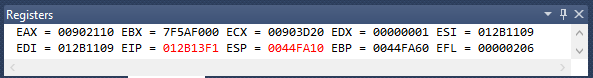
*\*\* NOTE -- Intel Processors store data in Little Endian Notation*

**Instance #2: Creating the Main() Stack Frame**

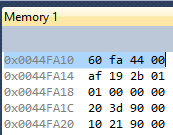
When the main function begins, a stack frame must be created in memory. This can be seen at the first instruction where a base pointer (ebp) is pushed into memory. This marks the beginning of the main stack frame, and does so by first setting up the stack pointer (esp).



*(Disassembly #2 - main())*



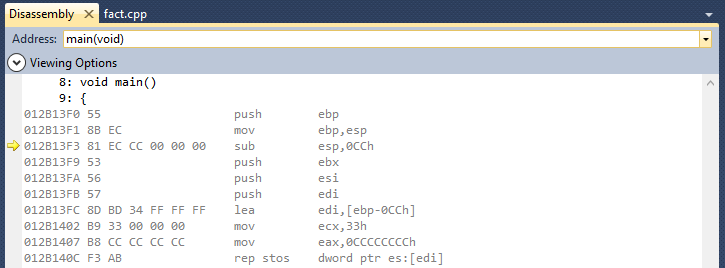
*(Registers #2)*

 *(Memory #2 - ESP)*

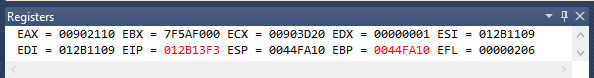
This instruction has set ESP equal to the address 0x0044FA10. It contains the address of the base pointer of the previous calling procedure, 0x0044FA60.

**Instance #3: Set Base Pointer and Stack Pointer of Main()**

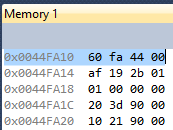
Next, the stack pointer (esp) is moved to the location of the base pointer (ebp), which is achieved by setting the base pointer equal to the stack pointer. The base pointer will always point to the base of the stack, however, the stack pointer will mark the top of the stack, and will grow accordingly as data is pushed and popped into and from the stack.



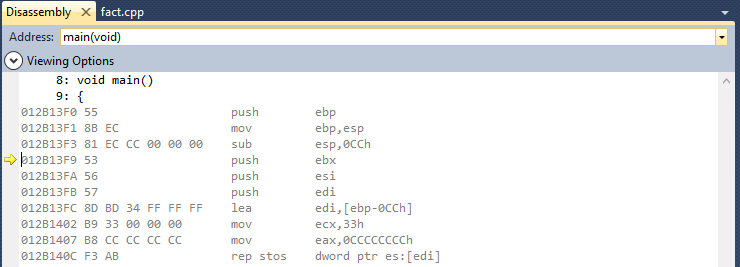
*(Disassembly #3 - main())*



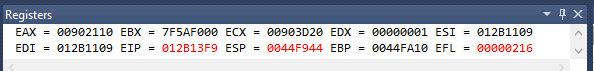
*(Register #3)*

 *(Memory #3 - EBP)*

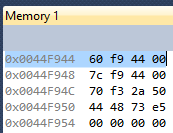
**Instance #4: Stack Space Allocation**



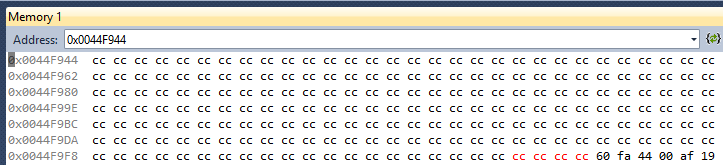
*(Disassembly #4 - main())*



*(Registers #4)*

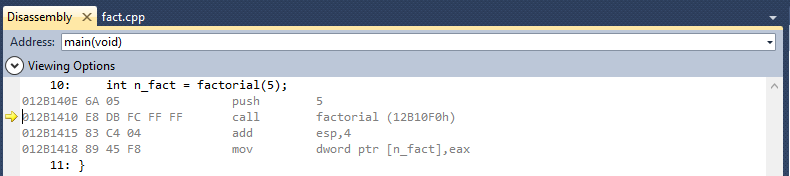
* (Memory #4 - ESP)*

The instruction "*81 EC CC 00 00 00 sub esp, 0CCh*"subtracts 204 bytes from the stack pointer, and stores the result by overwriting the stack pointer register. This can be seen from the instruction segment *CC 00 00 00*, or the segment *0CCh*, which represents the decimal number 204 in hexadecimal. What this is doing is moving the stack pointer and allocating 204 bytes between the stack pointer and the base pointer on stack. This brings the stack pointer to the new location 0x0044F944 and clears the allocated space over the next few instructions.

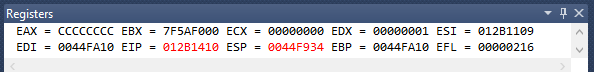


*(Memory #4 - ESP to EBP is cleared)*

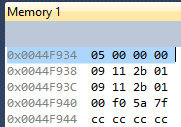
**Instance #5: Call and Jump to Factorial(5)**



*(Disassembly #5 - main())*

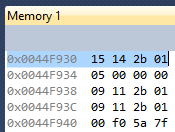


*(Registers #5)*

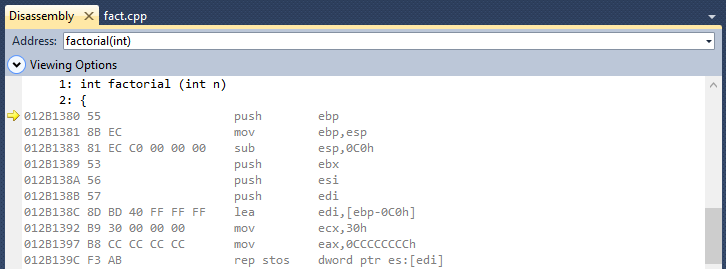
* (Memory #5 - ESP #1)*

This instruction pushes the parameter n = 5 to the stack, just before calling the factorial( int n ) function. The value can be seen at the location of the current stack pointer.

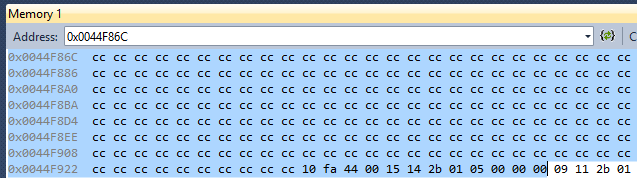
Immediately after, the call factorial procedure instruction is made and the program is then taken to the address 0x012B10F0. This address contains the instruction code to jump to the start of the factorial function in memory, which is at address 0x012B1380. In addition, the return address is stored in the address 0x0044F930 by the stack pointer, which contains 0x012B1415 -- the instruction immediately after the call factorial procedure.

 *(Memory #5 - ESP #2)*

**Instance #6: Factorial (5)**

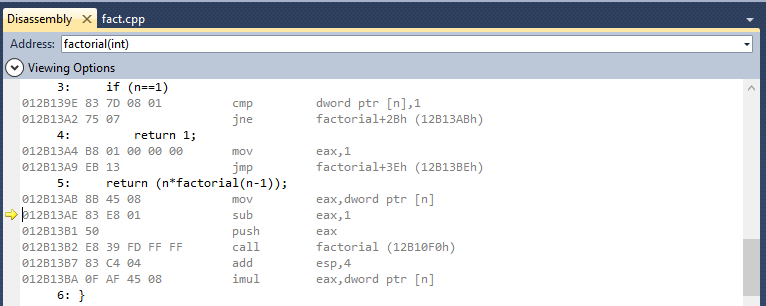


*(Disassembly #6 - factorial(5) #1)*

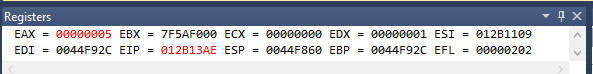


*(Memory #6 - factorial(5) stack frame)*

Now a stack frame must be set up for the factorial(5) procedure call. Similar to how the main() stack frame was setup, we can see the base pointer is initialized to address 0x0044F92C, 0C0h (192 bytes) are allocated and cleared in memory, and the stack pointer is currently set to address 0x0044F86C.

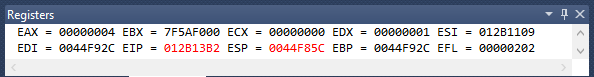


*(Disassembly #6 - factorial(5) #2)*

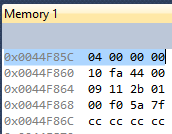


*(Registers #6 - EAX stores value 5)*

Here we can see the first conditional is ignored, and the program has jumped to the instruction at address 0x012B13AB. The value 5 is then stored into register EAX.

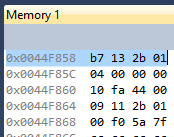


*(Registers #6 - EAX and ESP store value 4)*

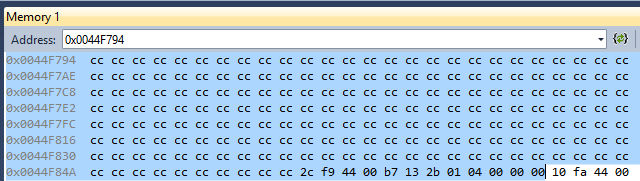
 *(Memory #6 - ESP store value 4)*

In the following instructions, 1 is subtracted from the value stored in EAX, then EAX is pushed onto the stack by the stack pointer at address 0x0044F85C. The program then recalls the factorial() procedure, and jumps back to the address of the first factorial() procedure instruction, 0x012B1380. This time, however, n is equal to 4.

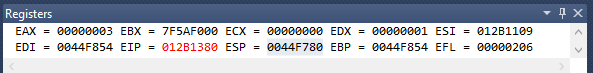
The return address is also stored at the address 0x0044F858, containing the address 0x012B13B7, which is the instruction immediately after the instruction that called the procedure.

*(Memory #6 - ESP stores return address)*

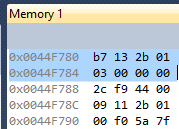
**Instance #7: Factorial(4)**

****

*(Memory #7 - factorial(4) stack frame)*



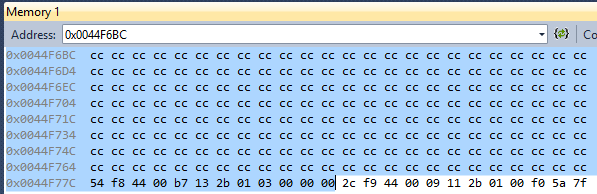
*(Registers #7)*

*(Memory #7 - return address and value 3)*

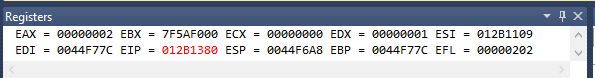
Similarly, a stack frame is created for the factorial(4) procedure call -- a base pointer is initialized to address 0x0044F854, n is decremented by 1, this value is then pushed onto the stack, the return address is stored, and the program recalls the factorial function once again.

One thing to notice is the return address for this frame stores the same instruction as the return address of the previous factorial(5) frame. This is because each instance of the recursive procedure is called by the same instruction, and will return to the instruction immediately afterwards.

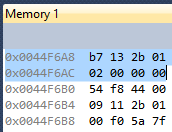
**Instance #8: Factorial(3)**



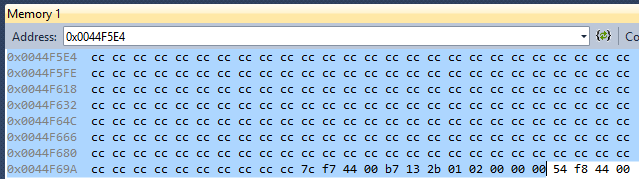
*(Memory #8 - factorial(3) stack frame)*



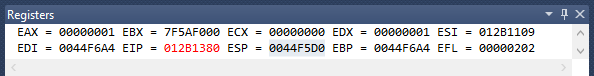
*(Registers #8)*

*(Memory #8 - return address and value 2)*

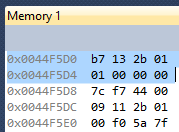
**Instance #9: Factorial(2)**



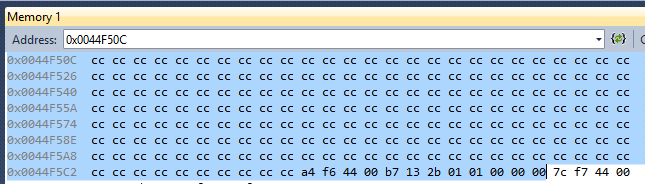
*(Memory #9 - factorial(2) stack frame)*

**

*(Registers #9)*

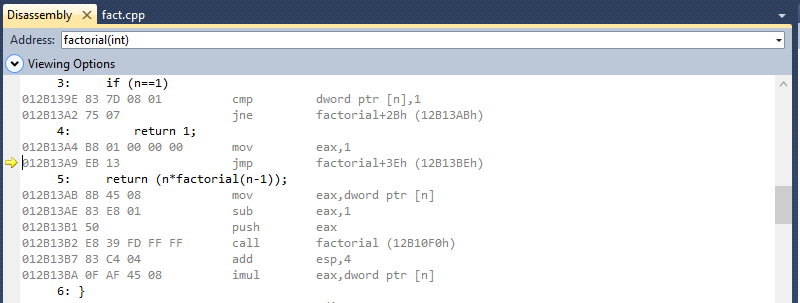
*(Memory #9 - return address and value 1)*

**Instance #10: Factorial(1)**

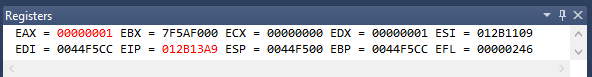


*(Memory #10 - factorial(1) stack frame)*

Now we have reached the last instance of the recursive procedure. This time, condition where n = 1 (the base case) will not be skipped, and is going to run. In this case, the condition simply returns the value 1. This value is then stored into register EAX.

**

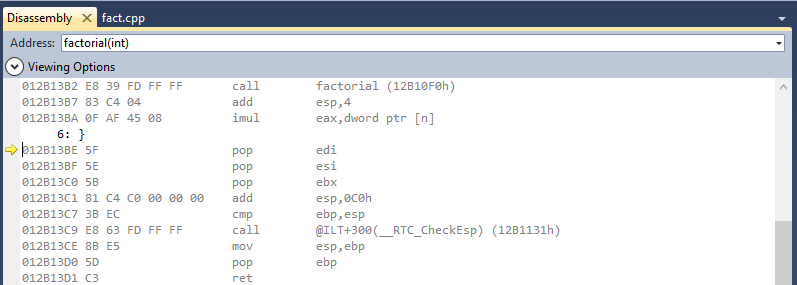
*(Disassembly #10 - factorial(1))*

**

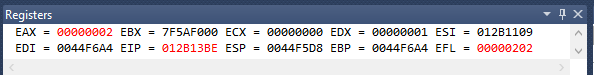
*(Registers #10 - EAX = 1)*

**Instance #11: Return and Evaluate n\_fact**

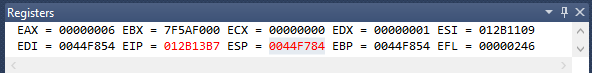
Next, the program jumps to the part of the code that will begin popping all of the factorial frames that are on stack. Each time, it multiplies our local variables cumulatively into register EAX. It will also continue to be called to the same return address (instruction 0x012B13B7) until all of the factorial stack frames are cleared.

**

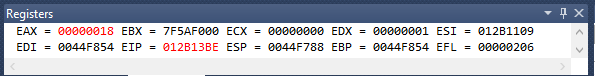
*(Disassembly #11 - factorial())*

**

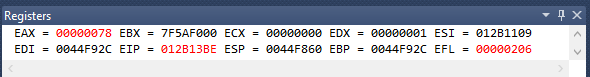
*(Registers #11 - EAX = 2)*

**

*(Registers #11 - EAX = 6)*

**

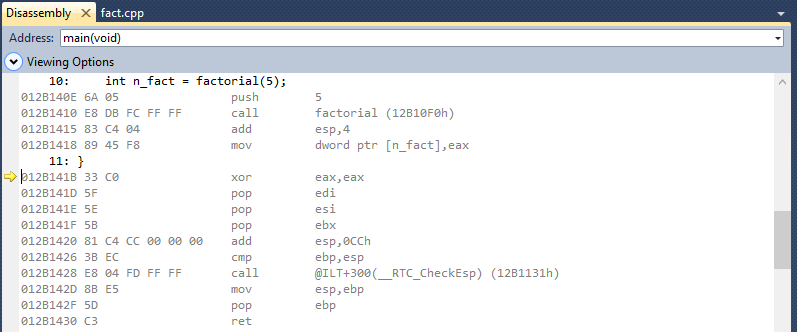
*(Registers #11 - EAX = 18)*

**

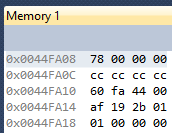
*(Registers #11 - EAX = 78)*

As we can see, EAX now equates to the hexadecimal value 0x00000078, or the decimal value 120, which is exactly what we expected. (5! = (5\*4\*3\*2\*1) = 120)

Lastly, we go back to main(), store the value of EAX into the variable n\_fact (address 0x0044FA08), and proceed to popping the main stack frame, and returning to the address after the main call procedure.



*(Disassembly #11 - main())*

*(Memory #11 - n\_fact = 0x00000078)*

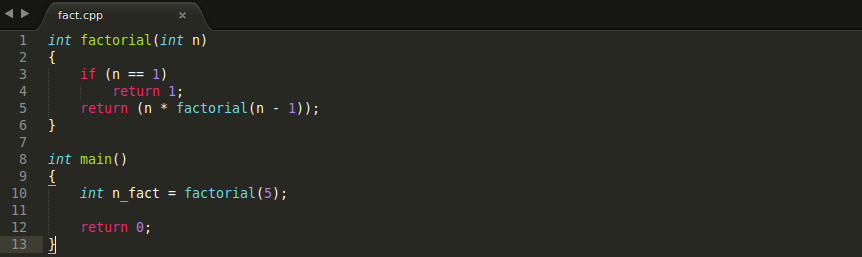
**Stack Visual**

|  |  |  |  |
| --- | --- | --- | --- |
| **Address** | **Content** |  |  |
| 0044FA60 | 0044FA68 | *EBP of calling* |  |
| ... | ... |  |  |
| 0044FA14 | 012B19AF | *Return Address* |  |
| 0044FA10 | 0044FA60 | *EBP of main()* | Main() |
| ... | ... |  |  |
| 0044FA08 | 00000078 | *n\_fact* |  |
| ... | ... |  |  |
| 0044F934 | 00000005 | *n = 5* |  |
| 0044F930 | 012B1415 | *Return Address* | Factorial(5) |
| 0044F92C | 0044FA10 | *EBP of fact(5)* |  |
| ... | ... |  |  |
| 0044F85C | 00000004 | *n = 4* |  |
| 0044F858 | 012B13B7 | *Return Address* | Factorial(4) |
| 0044F854 | 0044F92C | *EBP of fact(4)* |  |
| ... | ... |  |  |
| 0044F784 | 00000003 | *n = 3* |  |
| 0044F780 | 012B13B7 | *Return Address* | Factorial(3) |
| 0044F77C | 0044F854 | *EBP of fact(3)* |  |
| ... | ... |  |  |
| 0044F6AC | 00000002 | *n = 2* |  |
| 0044F6A8 | 012B13B7 | *Return Address* | Factorial(2) |
| 0044F6A4 | 0044F77C | *EBP of fact(2)* |  |
| ... | ... |  |  |
| 0044F5D4 | 00000001 | *n = 1* |  |
| 0044F5D0 | 012B13B7 | *Return Address* | Factorial(1) |
| 0044F5CC | 0044F6A4 | *EBP of fact(1)* |  |
| ... | ... |  |  |
| ... | ... | *ESP* |  |

*Stack Growth*

**Linux OS**

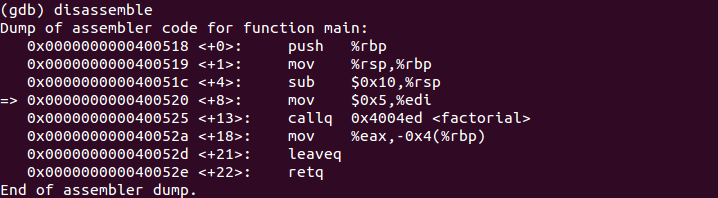
For Linux, we will be using Ubuntu to run GCC and GDB on a 64 bit AMD Phenom II X4 CPU. Let's first look at the following code segment, written in C:



*(Source Code in C)*

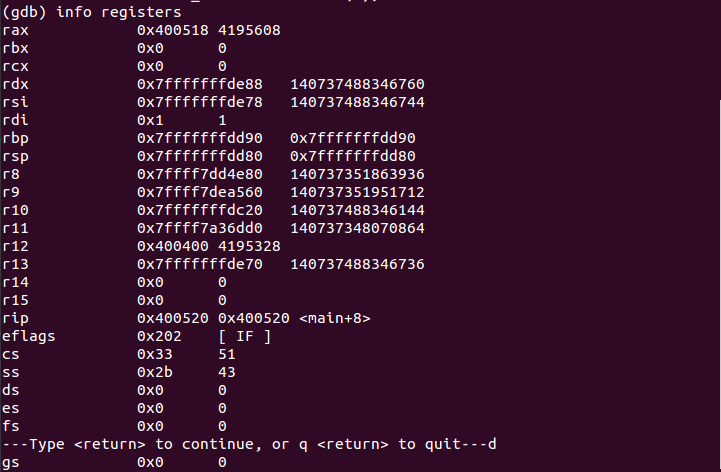
**Instance 1: Main Frame**

Now, if we begin debugging, we can set a break point at main(), and look at its disassembly:

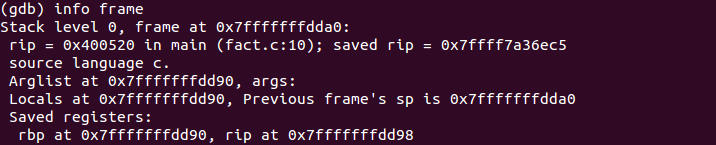


*(Disassembly - main())*

Here, we can see the main stack frame is created by first pushing the base pointer (rbp) to stack, moving the stack pointer (rsp) to the same location as the base pointer, and allocating 0x10 (16 bytes) for the frame.



*(Registers)*

**

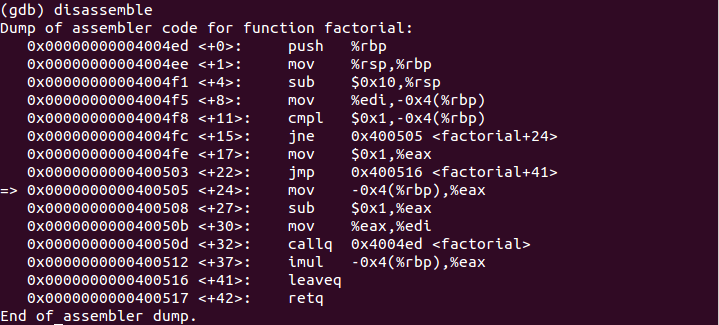
**

*(Info Frame)*

We can see the instruction pointer (rip) is at 0x400520, the base pointer is at 0x7FFFFFFFDD90, and the stack pointer is at 0x7FFFFFFFDD80.

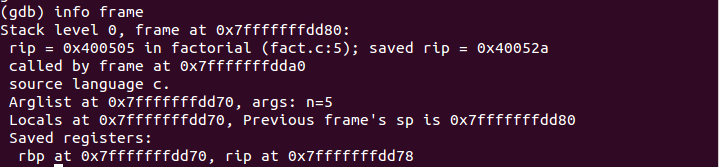
**Instance 2: Factorial(5) Frame**

Next, the program stores the parameter n = 5 into the register edi, followed by calling the factorial procedure. The program is then taken to the location in memory where the factorial procedure has been compiled.



*(Disassembly - factorial())*

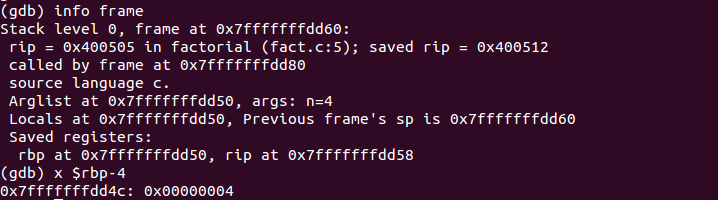
First, a stack frame is created for this procedure call, then, the value 5, stored in register edi, is moved into the memory location of the base pointer offset by 4 bytes (rbp - 4). The program then jumps to the else statement of our code, line 5. It will then move this value into register eax, subtract 1 from this value, and then move it back into register edi, just before recalling the factorial procedure on this new parameter.





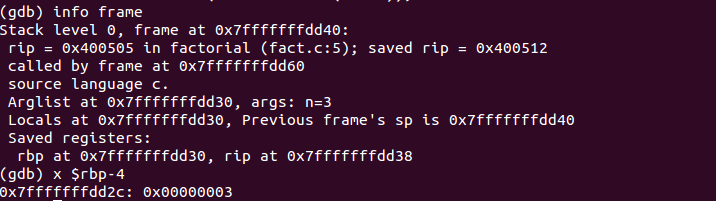
*(Info Frame and content of (rbp - 4))*

**Instance 3: Factorial(4) Frame**

****

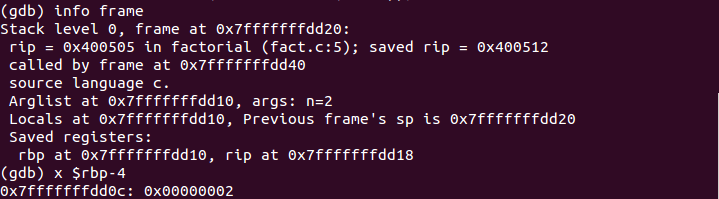
*(Info Frame)*

**Instance 4: Factorial(3) Frame**

****

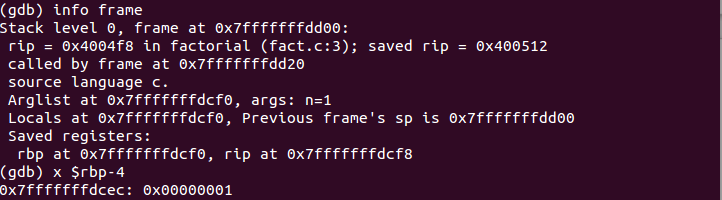
*(Info Frame)*

**Instance 5: Factorial(2) Frame**

****

*(Info Frame)*

**Instance 6: Factorial(1) Frame**

****

*(Info Frame)*

**Instance 7: Return**

The base case is finally met, and the factorial procedure returns 1, pops the frame from stack and jumps back to the previous frame. The program will continue to do this, at the same time begin to evaluate the product of all the stack arguments.











We get the value for 5! to be 120, or 0x78, which is what we expected. Finally, the program will return to the instruction in main right after the instruction that called the first factorial function, and proceed to pop the main function off stack.

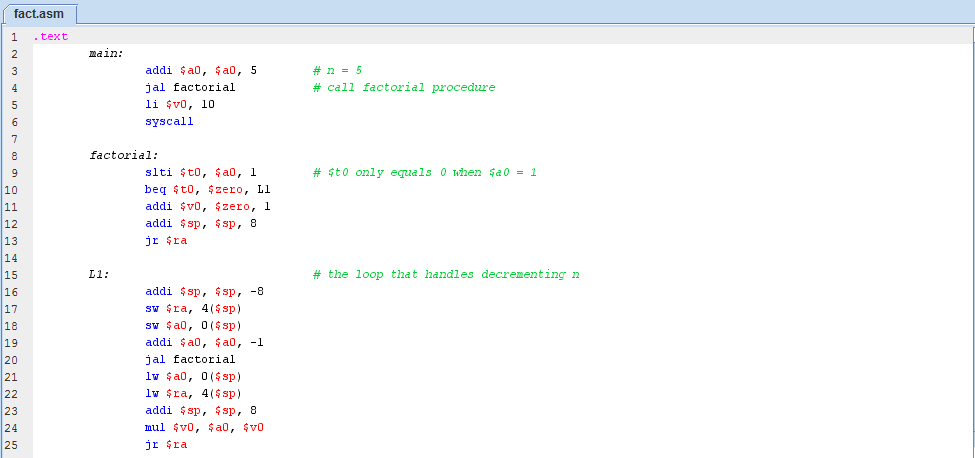
**Stack Visual**

|  |  |  |  |
| --- | --- | --- | --- |
| **Address** | **Content** |  |  |
| 0x7FFFFFFFDD90 | ... | *RBP of main()* |  |
| ... | ... |  | Main() |
| ... | 0x78 | *n\_fact* |  |
| ... | ... |  |  |
| 0x7FFFFFFFDD70 | 0x7FFFFFFFDD90 | *RBP of fact(5)* |  |
| 0x7FFFFFFFDD6C | 0x5 | *n = 5* | Factorial(5) |
| ... | ... |  |  |
| 0x7FFFFFFFDD50 | 0x7FFFFFFFDD70 | *RBP of fact(4)* |  |
| 0x7FFFFFFFDD4C | 0x4 | *n = 4* | Factorial(4) |
| ... | ... |  |  |
| 0x7FFFFFFFDD30 | 0x7FFFFFFFDD50 | *RBP of fact(3)* |  |
| 0x7FFFFFFFDD2C | 0x3 | *n = 3* | Factorial(3) |
| ... | ... |  |  |
| 0x7FFFFFFFDD10 | 0x7FFFFFFFDD30 | *RBP of fact(2)* |  |
| 0x7FFFFFFFDD0C | 0x2 | *n = 2* | Factorial(2) |
| ... | ... |  |  |
| 0x7FFFFFFFDCF0 | 0x7FFFFFFFDD10 | *RBP of fact(1)* |  |
| 0x7FFFFFFFDCEC | 0x1 | *n = 1* | Factorial(1) |
| ... | ... | *RSP* |  |

*Stack Growth*

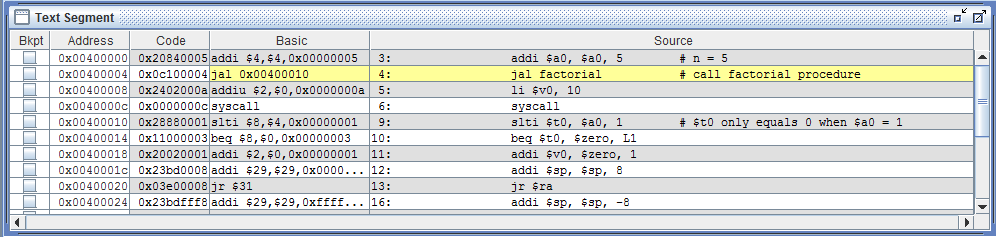
**MIPS/MARS**

Let's disassemble the following piece of code written in MIPS assembly:



*(MIPS Assembly Source Code)*

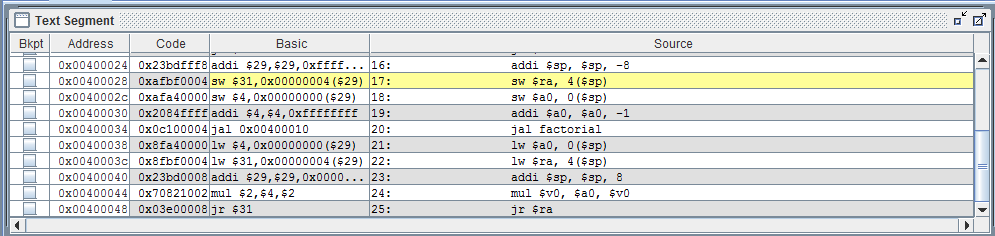
**Instance 1:**

****

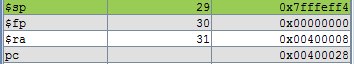
*(Text Segment - main)*

**** *(Register $a0 stores parameter n = 5)*

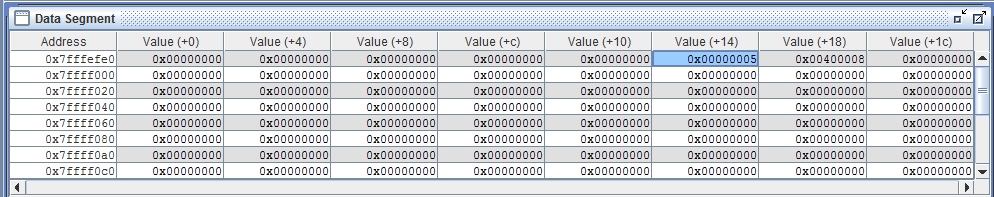
**Instance 2:**

****

*(Text Segment - L1)*

****

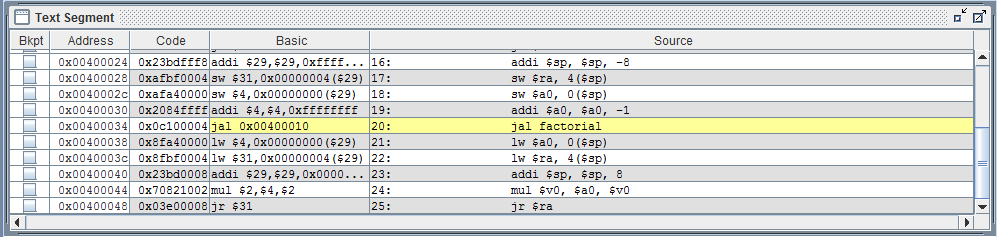
*(Stack pointer ($sp), return address ($ra), program pointer ($pc))*

****

*(Data Segment - return address and n = 5)*

Here we can see the value 5 has been stored into memory address 0x7FFFEFF4, and the return address for this procedure call in 0x7FFFEFF8.

**Instance 3:**

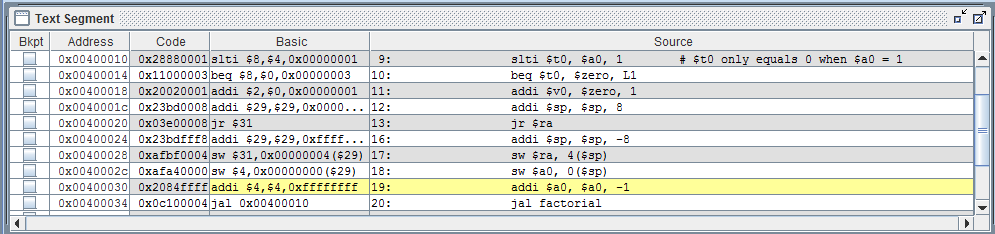


*(Text Segment - L1)*

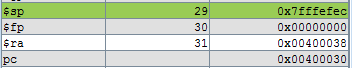
 *(Register $a0 stores parameter n = 4)*

The value n, stored in register $a0, has now been decremented by one.

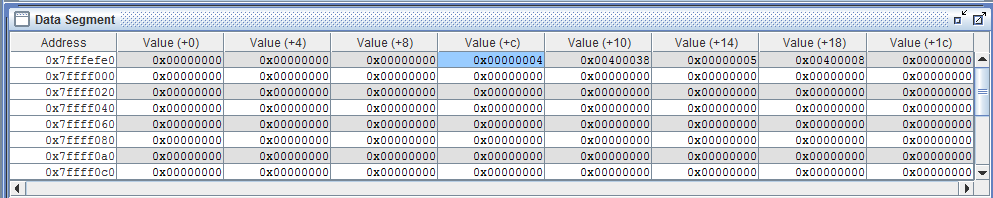
**Instance 4:**



*(Text Segment - L1)*

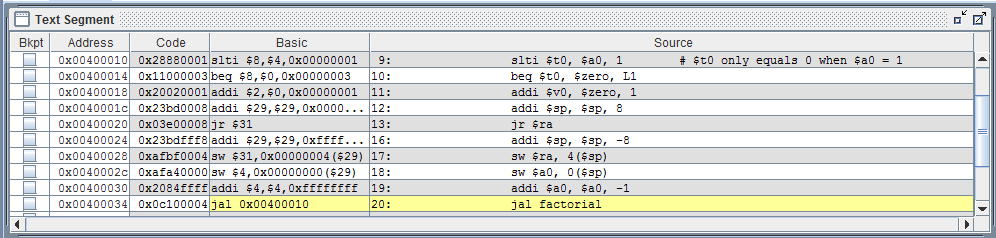


*Stack pointer ($sp), return address ($ra), program pointer ($pc))*



*(Data Segment - return address and n = 4)*

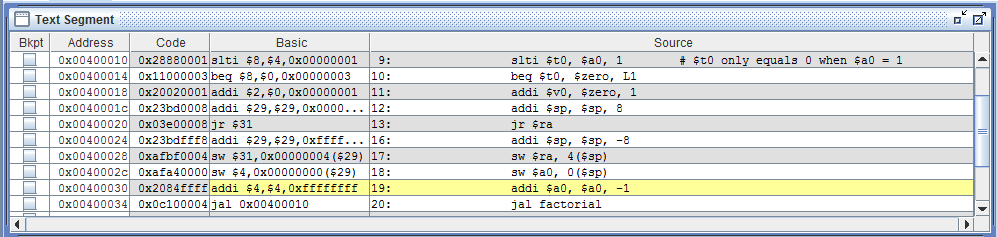
**Instance 5:**

****

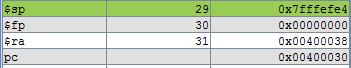
*(Text Segment - L1)*

**** *(Register $a0 stores parameter n = 3)*

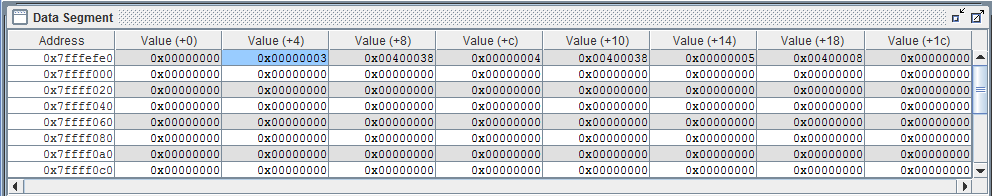
**Instance 6:**

****

*(Text Segment - L1)*

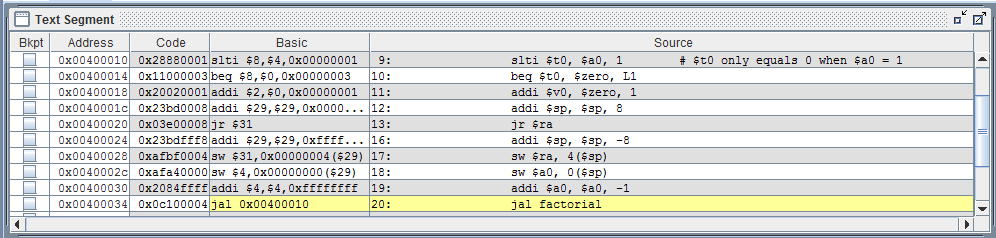
****

*(Stack pointer ($sp), return address ($ra), program pointer ($pc))*

****

*(Data Segment - return address and n = 3)*

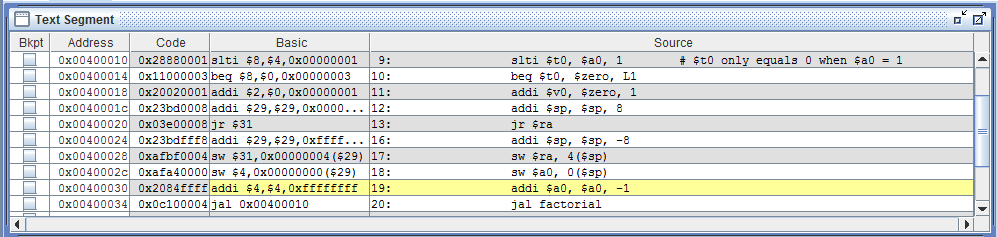
**Instance 7:**

****

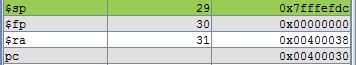
*(Text Segment - L1)*

* (Register $a0 stores parameter n = 2)*

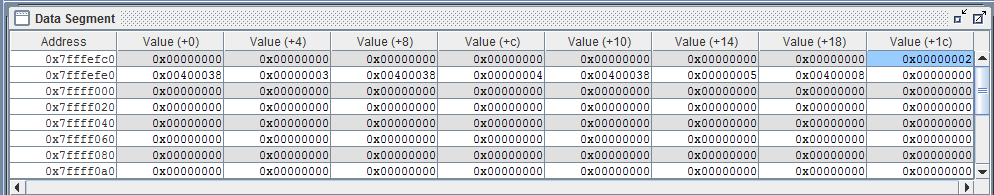
**Instance 8:**

****

*(Text Segment - L1)*

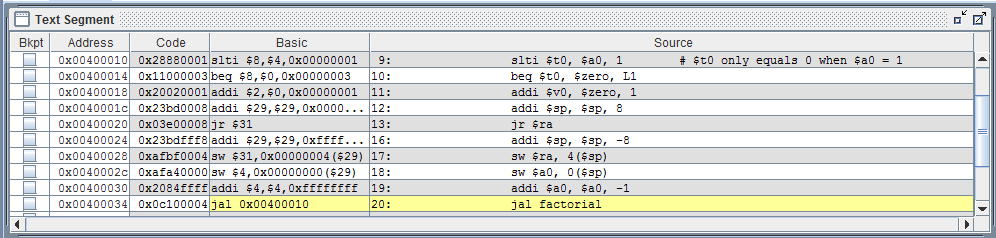
****

*(Stack pointer ($sp), return address ($ra), program pointer ($pc))*

****

*(Data Segment - return address and n = 2)*

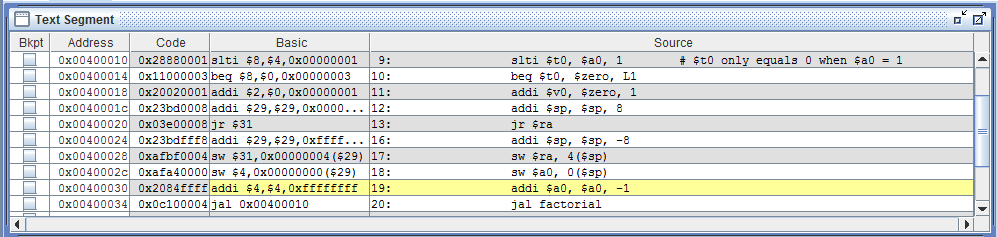
**Instance 9:**

****

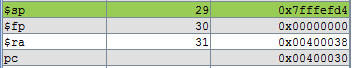
*(Text Segment - L1)*

* (Register $a0 stores parameter n = 1)*

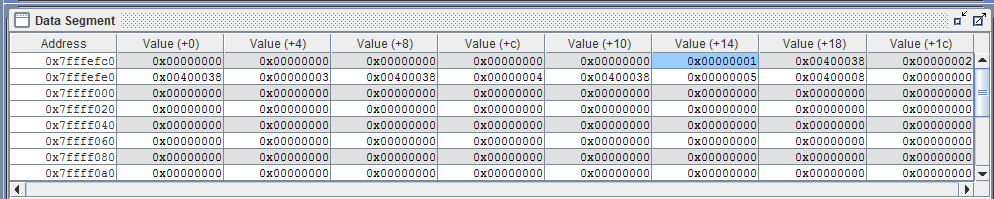
**Instance 10:**

****

*(Text Segment - L1)*

****

*(Stack pointer ($sp), return address ($ra), program pointer ($pc))*

****

*(Data Segment - return address and n = 1)*

**Instance 11:**

Now the stack frames must be popped and the values must multiplied.

****

****

****

****

Once again, we get the end result of 0x78, or 120.

**Stack Visual**

|  |  |  |  |
| --- | --- | --- | --- |
| **Address** | **Content** |  |  |
| 0x7FFFEFF8 | 0x00400008 | *return address* | Factorial(5) |
| 0x7FFFEFF4 | 0x00000005 | *n = 5* |  |
| 0x7FFFEFF0 | 0x00400038 | *return address* | Factorial(4) |
| 0x7FFFEFEC | 0x00000004 | *n = 4* |  |
| 0x7FFFEFE8 | 0x00400038 | *return address* | Factorial(3) |
| 0x7FFFEFE4 | 0x00000003 | *n = 3* |  |
| 0x7FFFEFE0 | 0x00400038 | *return address* | Factorial(2) |
| 0x7FFFEFDC | 0x00000002 | *n = 2* |  |
| 0x7FFFEFD8 | 0x00400038 | *return address* | Factorial(1) |
| 0x7FFFEFD4 | 0x00000001 | *n = 1* |  |

*Stack Growth*

**Conclusion:**

We have used three different platforms to analyze how a recursive factorial procedure is interpreted by the CPU and stored in memory. We saw how stack frames must be created in memory for each instance the procedure is recalled. At each instance, the argument of n is decremented by one and stored in memory. Stack frames are then destroyed once the recursive base case is met, and the program links back to the previous frame via the return address. During this process, the arguments at each frame level are multiplied to give us the final result of n factorial. This summarizes how the CPU handles recursive procedure calls.