**Assignment No.9**

|  |
| --- |
| **Title of Assignment:**  Write x86 ALP to find the factorial of a given integer number on a command line by using  recursion. Explicit stack manipulation is expected in the code.. |
| **Relevant Theory:**  **Design Analysis/ Implementation Logic:**  **Stack Frame:**  The *stack frame*, also known as *activation record* is the collection of all data on the stack associated with one subprogram call. The stack frame generally includes the following components:   * The return address * Argument variables passed on the stack * Local variables (in HLLs) * Saved copies of any registers modified by the subprogram that need to be restored.   Since the stack frame for a given subprogram has a fixed size, we can reduce the number of instructions required to push and pop it by only updating the stack pointer once. Instead of individual pushes and pops, we can use offset addressing: The Frame Pointer The stack pointer will change when a subprogram does a push or pop operation. Many subprograms do this during calculations for convenience, or to implement algorithms that use a stack. When this happens, the offset addresses representing local automatic variables such as 4($sp) are no longer valid. The offsets are computed by the compiler, and hard-coded as offset-mode operands in the instructions, so they cannot be easily changed while the program is running. One way to alleviate this problem is by using the frame pointer. The frame pointer is another register that we set to the address of the stack frame when a subprogram begins executing. If the code refers to local variables as offsets from the frame pointer instead of offsets from the stack pointer, then the program can use the stack pointer without complicating access to auto variables. We would then refer to something in the stack frame as offset($fp) instead of offset($sp).  Address Content  $fp --> F080 Saved return address  F084 a  F088 b  F090 x  F094 y  F098 Saved register value  F09C Saved register value       Passing Arguments on the Stack In much architecture, subprogram arguments are by convention passed on the system stack rather than in registers. The use of registers for arguments is only common in architectures that have many general-purpose registers available. Even in architectures that have a large number of registers such as the MIPS, there are often subprograms that take many arguments, making it impractical to pass all of them in registers.  **Algorithm**   1. Pop the stack & store the value in ECX register. 2. Move this value in ECX to argc variable.. 3. Compare argc with 2 if not equal print error and goto step 4. Pop the stack to ignore file name. 5. Pop the stack again to get base address of first command line parameter i.e. number. 6. Read it from the base address character by character till null character in varaiable num 7. Push num 8. Call factorial 9. Increase the stack pointer by 8 to get return value in rax 10. Copy return value in rbx 11. Call display 12. End   Factorial Function Algorithm:  1 Push rbp  2. Copy rsp in rbp as we do not want to modify rsp.  3. Get the number in rbx using offset rbp+16  4. compare rbx with 1 if equal goto step 11  5. dec rbx  6. Push rbx  7. Call factorial  8. add 8 in rsp  9. Get the number in rbx using rsp+16  10 Multiply rax and rbx  11.copy rbp in rsi  12 pop rbp  13 ret |
| **Testing:**  **Test Conditions:**  **Input:**  Command line Input : Number  **Output:**  Factorial of the number (n\*(n-1)\*(n-2)\*…..3\*2\*1) |
| **FAQs:**   1. Explain Recursion. 2. Explain Stack manipulation in recursion. 3. Explain the Program stack. |
| **Conclusion:**  Successfully implemented the ALP to find the factorial of a given integer number on a command line by using recursion with explicit stack manipulations. |