Lab 7 – NIOS II Stack and Function calls

**Deliverables & Questions Worksheet –** *See instructions for context*

**Name & Date & Section:**

Jacob Brink

10/18/2019

Section A

**D1: C-Code for Factorial Function**

/\*

\* Engr220L - Lab 7 Prelab

\* Date: 10/18/2019

\* Author: Jacob Brink

\*

\* This file defines the factorial function

\* previously declared in main.c. It recursively

\* calculates the factorial of the received input value.

\*

\*/

int factorial (int n)

{

if (n > 1)

return n \* factorial(n-1);

return 1;

}

**Q2: “How does it know when to break out of the loop?”**

It knows to break out of the loop when it compares r16 and r18, r18 is the exclusive end of the loop range at 11. When r16 is greater than r18 or equal to it breaks.

**D3: Main Loop Table**

|  |  |  |
| --- | --- | --- |
| **Main() Iteration Loop Count (“n”)** | **R4 Value Before “Call Factorial”** | **R2 Value After “Call Factorial”** |
| 0 | 0 | 1 |
| 1 | 1 | 1 |
| 2 | 2 | 2 |
| 3 | 3 | 6 |
| 4 | 4 | 24 |
| 5 | 5 | 120 |
| 6 | 6 | 720 |
| 7 | 7 | 5040 |
| 8 | 8 | 40320 |
| 9 | 9 | 362880 |
| 10 | 10 | 3628800 |
|  |  |  |

**Q4: “Describe in words how registers 2 and 4 are being used in the main routine.”**

Register 2 is used for the return value of the function, which is the factorial of Register 4, which is the loop counter.

**D5: Recursion Analysis Table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Breakpoint Count** | **Value of n in Main()** | **Location of Breakpoint (one per row)** | | | |
| **R4 Value at main() call statement** | **R2 Value After main() call statement** | **R4 Value at factorial() call statement** | **R2 Value After factorial() call statement** |
| 1 | 0 | 0 |  |  |  |
| 2 |  |  | 1 |  |  |
| 3 |  | 1 |  |  |  |
| 4 |  |  | 1 |  |  |
| 5 |  | 2 |  |  |  |
| 6 |  |  |  | 1 |  |
| 10 |  |  |  |  | 1 |
| 11 |  |  | 2 |  |  |
| 12 |  | 3 |  |  |  |
| 13 |  |  |  | 2 |  |
| 14 |  |  |  | 1 | 1 |
| 15 |  |  |  |  | 2 |
| 16 |  |  | 6 |  |  |
| 17 |  | 4 |  |  |  |
| 18 |  |  |  | 3 |  |
| 19 |  |  |  | 2 |  |
| 20 |  |  |  | 1 |  |

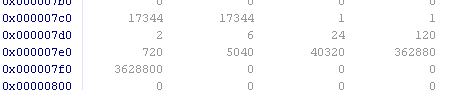
**Q6: “Comment on the sequence of events listed in the table and what they represent.”**

When the factorial function is called, R4 is used to pass the function argument to the factorial function. R4 then is used again when the function is recursively called. This time the recursive function calls with a parameter n-1 so R4 decreases until the base case is called (when n == 1) and then the functions begin to return. This sets off a similar chain of events in which R2 represents the return of each function. R2 is first 1, because that is the base case, then it is 2, then 6, then 24 and so on and so on until it is n!

**D7: Global Array Table**

|  |  |  |
| --- | --- | --- |
| **Main Loop Iteration (“n”)** | **Effective Address Used by “stw” (use hexadecimal)** | **Data to be stored at that address (use decimal)** |
| 0 | 0x000007C8 | 1 |
| 1 | 0x000007CC | 1 |
| 2 | 0x000007D0 | 2 |
| 3 | 0x000007D4 | 6 |
| 4 | 0x000007D8 | 24 |
| 5 | 0x000007DC | 120 |
| 6 | 0x000007E0 | 720 |
| 7 | 0x000007E4 | 5040 |
| 8 | 0x000007E8 | 40320 |
| 9 | 0x000007EC | 362880 |
| 10 | 0x000007F0 | 3628800 |

**D8: Factorial Array Memory Screenshot**



**Q9: “How many 4-byte (1-word) elements are currently on the stack?”**

6

**Q10: “Identify the address pointed to by the stack pointer.”**

0x00007fdc

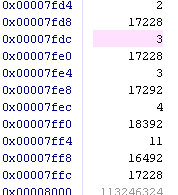
**Q11: “What is the state of the stack now?”**

skipped

**Q12: “How many items are on the stack?”**

9

**D13: Values on the Stack Screenshot**

****

**Q14: “Write comments that identify what is stored in each location of the stack.”**

Each location on the stack represents either the n each recursive call gets or the address of the next instruction to be called. For example, if n is 5, when the recursion goes down all the way to the bottom, the stack will be populated with 5, 4, 3, and 2 with 2 at the top of the stack. Then, in between each number is the address of the next instruction, which in this case is the multiplication of the current top item in the stack to the last popped item in the stack.

When the factorial function is called, it stores the value of r16, which is n, of the caller, before setting its value to the given n, which in this case would be n-1. It stores r16 by adding it to the stack and adding the multiply instruction on the stack as well. When it reaches the base case, it simply pops off value, multiplies popped with r2, which is the “accumulator” until n! is reached.