ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 351E

MICROCOMPUTER LABORATORY EXPERIMENT REPORT

EXPERIMENT NO	:	4
EXPERIMENT DATE	:	06.12.2024
LAB SESSION	:	FRIDAY - 14.30
GROUP NO	:	G9

GROUP MEMBERS:

150220723 : Abdulsamet Ekinci

150210906 : Emil Huseynov

150220902 : Nahid Aliyev

FALL 2024

Contents

1 INTRODUCTION [10 points]

Understanding low-level operations such as function calls and stack management is crucial for optimizing software performance and ensuring efficient memory usage. In this experiment, we focused on implementing and analyzing advanced assembly programming techniques using the MSP430 microcontroller. Specifically, we implemented the Russian Peasant Division (RPD) algorithm for modulus calculation, a hashing algorithm with linear probing for collision resolution, and a recursive dot product subroutine. The objectives were to explore complex low-level operations, handle collisions in hash tables, and manage recursion through efficient stack utilization.

2 MATERIALS AND METHODS [40 points]

This section describes the hardware, software, and methods used in the experiment.

2.1 Hardware

- **MSP430G2553 Microcontroller:** A low-power microcontroller with 16-bit RISC architecture.
- **LEDs:** Connected to the output pins for displaying the results of the calculations.
- **Buttons:** Configured as inputs for triggering actions based on conditions.

2.2 Software

- **Code Composer Studio (CCS):** Used to write, debug, and flash assembly programs onto the MSP430.

2.3 Procedure

The experiment was divided into three main parts, each focusing on a specific aspect of assembly programming and stack management using the MSP430 microcontroller.

2.3.1 Part 1: Function Calls and Stack Operations

Objective: Implement and analyze function calls and stack operations to understand how the stack is utilized during program execution.

Procedure:

- **Step 1: Setup** - Reserve space for the result array and initialize the input array.

- Memory allocation is performed to reserve space for the result array using the '.bss' directive, ensuring that sufficient memory is allocated without initializing its content.
- The input array is initialized with a set of predefined 8-bit integers. This setup prepares the data that will be processed during the program's execution.
- The address of the last element in the array is stored to control the termination condition of the main processing loop.

- **Step 2: Execute and Monitor** - Run the program on the MSP430 and use the CCS debugger to observe register values and stack states during function calls.
  - The 'main' function initializes registers with the base addresses of the input and result arrays.
  - The program enters a loop where it processes each element of the input array by calling 'func1'. This function, in turn, calls 'func2', demonstrating nested function calls and their impact on the stack.
  - Using the CCS debugger, monitor how each function call affects the stack pointer (SP) and other registers, ensuring that return addresses and parameters are correctly managed.

2.3.2 Part 2: Hashing with Linear Probing

Objective: Implement a hashing algorithm using linear probing to resolve collisions and store student ID values efficiently.

Procedure:

- **Step 1: Initialize the Hash Table**
  - The hash table is set up in memory, with each slot initialized to a default value (e.g., '-1') to indicate that it is empty.
  - Basic arithmetic subroutines such as 'Add', 'Subtract', and 'Multiply' are developed to support operations required by the hashing algorithm, such as calculating hash values and handling collisions.

- **Step 2: Implement Add and Subtract Subroutines**
  - Subroutines for addition and subtraction are created to handle arithmetic operations necessary for hashing. These subroutines manage parameter passing via the stack, ensuring that inputs are correctly retrieved and results are properly returned.

- Care is taken to maintain stack integrity by matching each ‘push‘ operation with a corresponding ‘pop‘, preventing stack corruption.

- **Step 3: Implement Multiply Subroutine**
  - A multiplication subroutine is implemented using a loop that performs repeated addition, simulating the multiplication process. This approach demonstrates how more complex arithmetic operations can be managed at the assembly level.
  - The subroutine includes checks for edge cases, such as multiplying by zero, to ensure robustness and accurate results.

2.3.3 Part 3: Recursive Dot Product Subroutine

Objective: Implement a recursive subroutine to compute the dot product of two vectors, emphasizing stack management during recursive calls.

Procedure:

- **Step 1: Initialize the Stack and Parameters**
  - The stack pointer (SP) is initialized to a designated memory address to manage stack operations efficiently.
  - Base addresses of the input arrays (‘arrayA‘ and ‘arrayB‘) are loaded into registers and pushed onto the stack.
  - Initialization of the index ‘i‘ and the length ‘N‘ of the arrays is performed, with these values also pushed onto the stack to serve as parameters for the recursive subroutine.

- **Step 2: Execute and Monitor** - Run the program on the MSP430 and use the CCS debugger to observe the behavior of recursive calls and how the stack is managed during the dot product calculation.
  - Each recursive call to ‘dotProduct‘ pushes new parameters onto the stack, including updated pointers and indices.
  - The base case is monitored to ensure that recursion terminates correctly when the index ‘i‘ reaches the length ‘N‘ of the arrays.
  - Stack integrity is crucial to prevent overflow and ensure accurate aggregation of results from recursive calls. Observations are made on how each call and return affects the stack and registers.

3 RESULTS AND DISCUSSION [30 points]

Both the Russian Peasant Division (RPD) algorithm and the hashing algorithm with linear probing were successfully implemented and tested on the MSP430 microcontroller. Additionally, the recursive dot product subroutine was implemented to compute the dot product of two vectors. The results were displayed using LEDs connected to Port 1, and the hash table was populated with hashed student ID values.

3.1 Part 1: Function Calls and Stack Operations

For Part 1, after running the program and using the CCS debugger, we recorded the register values and stack contents at various points during the execution of the main loop and function calls. The table below reflects a snapshot of these results.

Table 1: Register and Stack State During Function Call (Part 1)

Code	PC	R5	R10	R6	R7	SP	Content of the Stack
`mov #array, r5`	c012	200	-	-	-	-	-
`mov #resultArray, r10`	c016	200	206	-	-	-	-
`mov.b @r5, r6`	c018	200	206	1	-	-	-
`inc r5`	c01A	201	206	1	-	-	-
`call #func1`	c02C	201	206	1	-	03FE	c01C
`dec.b r6`	c02E	201	206	0	-	03FE	c01C
`mov.b r6, r7`	c030	201	206	0	0	03FE	c01C
`call #func2`	c038	201	206	0	0	03FC	c01C, c032
`xor.b #0FFh, r7`	c03A	201	206	0	00ff	03FC	c01C, c033
`ret`	c034	201	206	0	-	03FE	c01C
`mov.b r7, r6`	c036	201	206	00ff	-	03FE	c01C
`ret`	c01E	201	206	-	-	400	-
`mov.b r6, 0(r10)`	c022	201	206	-	-	400	-
`inc r10`	c024	201	207	-	-	400	-

Analysis: The observation confirmed that each function call placed a return address on the stack and that parameters were handled consistently. The subtle changes in the Stack Pointer (SP) and the stack content underscore how each call and return modifies the runtime state. Specifically, calling ‘func1‘ decreased the SP by 2 bytes (assuming a word-sized stack), and returning from ‘func1‘ restored the SP to its previous state.

3.2 Part 2: Hashing with Linear Probing

The implementation of the hashing subroutines demonstrated the use of the stack for parameter passing and result retrieval. The ‘Add‘, ‘Subtract‘, and ‘Multiply‘ subroutines effectively utilized the stack to manage inputs and outputs.

Addition Subroutine: The ‘Add‘ subroutine successfully added two numbers passed via the stack and returned the result. Similarly, the ‘Subtract‘ and ‘Multiply‘ subroutines performed their respective operations correctly, showcasing the stack's role in function parameter management.

Challenges and Observations:

- Ensuring correct stack manipulation during parameter passing was critical. Incorrect stack operations could lead to corrupted data or unexpected behavior.
- Handling edge cases, such as negative numbers in the multiplication subroutine, required careful attention to ensure accurate results.

3.3 Part 3: Recursive Dot Product Subroutine

The recursive dot product subroutine aimed to compute the dot product of two vectors by recursively multiplying corresponding elements and summing the results.

Results: While the subroutine successfully computed the dot product for some inputs, issues were encountered during deeper recursion levels, leading to stack overflow or incorrect results.

Challenges and Observations:

- **Stack Trace Problems:** The recursive calls were not properly unwinding the stack, causing the stack to grow beyond its allocated space. This was primarily due to incorrect push and pop operations, leading to stack corruption.
- **Parameter Management:** Passing and retrieving parameters in recursive calls were mishandled, resulting in incorrect computation of the dot product. Specifically, the increment operations on the array pointers and index ‘i‘ were not consistently managed across recursive calls.
- **Base Case Handling:** The base case was correctly identified when the index ‘i‘ reached the length of the arrays (‘N‘). However, the return mechanism did not correctly aggregate the results from recursive calls, leading to inaccurate final results.

Stack Trace Analysis: During the execution of the recursive subroutine, each call to ‘dotProduct‘ should have correctly pushed the current state onto the stack and retrieved it

upon returning. However, due to mismanagement in the push/pop sequence and incorrect register usage, the stack became inconsistent. This led to the overwriting of crucial data and eventual stack overflow.

Solutions Attempted:

- **Reviewing Push/Pop Operations:** Ensured that every ‘push‘ operation had a corresponding ‘pop‘ to maintain stack balance.
- **Parameter Passing Correction:** Verified that parameters were correctly passed and retrieved in each recursive call.
- **Debugging with CCS:** Used the CCS debugger to step through each recursive call, monitoring the stack pointer and register states to identify inconsistencies.

Despite these efforts, fully resolving the stack trace issues required a more in-depth revision of the recursive subroutine's logic, ensuring accurate parameter management and stack operations.

4 CONCLUSION [20 points]

This experiment deepened our understanding of low-level programming concepts such as function calls, stack management, and recursion using the MSP430 microcontroller. We successfully implemented the Russian Peasant Division (RPD) algorithm for modulus calculation and a hashing algorithm with linear probing for collision resolution. Additionally, we attempted to implement a recursive dot product subroutine, which highlighted the complexities involved in managing recursion and stack operations at the assembly level.

Key Learnings:

- Effective use of the stack is essential for managing function calls and parameter passing in assembly programming.
- Implementing recursive algorithms requires meticulous stack management to prevent issues like stack overflow and data corruption.
- Debugging low-level code necessitates a thorough understanding of the microcontroller's architecture and the stack's behavior.