**EE2024 Programming for Computer Interfaces**

**Assignment 1 Report**

|  |  |
| --- | --- |
| **Name: Ang Kah Min, Kelvin** | **Matric No: A0111890** |
| **Name: Sitti Maryam Binte Rashid Ridza** | **Matric No: A0112675** |

# Lab Manual Questions

**Describe what happens and explain why.**

The execution proceeds into distrBF(…), runs the ELEMENT subroutine twice, before looping indefinitely on line 19 (BX LR; the instruction to return to the address in the Link Register).

In a nutshell, this is because the Link Register (LR) does not contain the correct address to return back to the main program. The address in the LR (which was pointing to the start of the for-loop in the main program) was overwritten when the ELEMENT subroutine was called and the line repeatedly “returns” to the instruction after the last ELEMENT subroutine call (which is line 19).

To supplement our explanation, we have provided a table with relevant addresses for the instructions:

|  |  |  |
| --- | --- | --- |
|  | Address | Instruction |
| Main C Program | 0x000001C2 | int dij[T][N]=…; |
| 0x000001E0 | int Dj[T][N]=…; |
| 0x000001FE | int n\_res[T][2]=…; |
| 0x00000222 | distrBF(T,(int\*)n\_res,(int\*)dij,(int\*)Dj); |
| 0x00000237 | for ( i=0; i<T ; i++ ) … |
|  | … |  |
| distrBF: | 0x000001AC | BL ELEMENT |
| 0x000001B0 | BL ELEMENT |
| 0x000001B4 | BX LR |
|  | … |  |
| ELEMENT: | 0x000001B6 | BX LR |

To explore this problem in detail, we will now break down the contents of the LR and PC in four steps:

1) When the distrBF(…) function is first called (at address 0x0000222), the address in the PC (which has already incremented to 0x00000237 after the atomic operation) is stored in the Link Register. This is the address to be used to return to the main program distrBF(…) has finished executing. The PC is then loaded with distrBF’s address (0x000001AC) before the next Fetch & Execution (F&E) cycle.

|  |  |
| --- | --- |
| **Link Register** | **Program Counter** |
| 0x00000237 | 0x000001AC |

(Note that LR values in the actual hardware is always 1 higher than the one shown in this report. This is because the 1 bit is set to signify that the return instruction is a THUMB instruction).

2) When ELEMENT is first called (at address 0x000001AC), it also stores the address in the PC (which has already incremented to 0x000001B0 after the fetch cycle) in the LR. The address back to the main program gets overwritten here (and it cannot be retrieved anymore). The PC is loaded with the address specified by the ELEMENT label (0x000001B6) for the next F&E cycle.

|  |  |
| --- | --- |
| **Link Register** | **Program Counter** |
| 0x000001B0 | 0x000001B6 |

ELEMENT only contains BX LR, which returns to the address specified in the LR.

|  |  |
| --- | --- |
| **Link Register** | **Program Counter** |
| 0x000001B0 | 0x000001B0 |

3) When ELEMENT is called the second time (at address 0x000001B0), the same thing occurs again as with the first time. Except that the LR is now loaded with address 0x000001B4.

|  |  |
| --- | --- |
| **Link Register** | **Program Counter** |
| 0x000001B4 | 0x000001B6 |

Once again, BX LR causes the PC to be replaced with the address in the LR.

|  |  |
| --- | --- |
| **Link Register** | **Program Counter** |
| 0x000001B4 | 0x000001B4 |

4) When distrBF(…) tries to return to the C program (at address 0x000001B4), it attempts to load the address in the LR into the PC. However, it is unsuccessful because the value in the LR is still 0x000001B4 instead of 0x00000237 as expected. This causes the execution to loop indefinitely at this line.

**Explain the different behavior of the program.**

The console now prints the contents of the n\_res array as shown:

4 0

0 0

0 0

We will now refer to the following updated address table:

|  |  |  |
| --- | --- | --- |
|  | Address | Instruction |
| Main C Program | 0x000001C6 | int dij[T][N]=…; |
| 0x000001E4 | int Dj[T][N]=…; |
| 0x00000202 | int n\_res[T][2]=…; |
| 0x00000226 | distrBF(T,(int\*)n\_res,(int\*)dij,(int\*)Dj); |
| 0x0000023A | for ( i=0; i<T ; i++ ) … |
|  | … |  |
| distrBF: | 0x000001AC | PUSH {R14} |
| 0x000001AE | BL ELEMENT |
| 0x000001B2 | BL ELEMENT |
| 0x000001B6 | POP {R14} |
| 0x000001BA | BX LR |
|  | … |  |
| ELEMENT: | 0x000001BC | BX LR |

As before, we will now explain this detail in four phases, but I will omit details previously mentioned:

1) When the distrBF(…) function is first called (at address 0x0000226), the LR and PC is changed as before. When PUSH {14} is executed, it decrements the stack pointer (because it is a full decrementing stack) by 4 bytes (from 0x10007F60 to 0x10007F5C) and stores the contents of R14 (the LR) there.

|  |  |
| --- | --- |
| **Link Register** | **Program Counter** |
| 0x0000023A | 0x000001AE |

|  |  |
| --- | --- |
| **Stack Address** | **Value** |
| 0x10007F60 | 0x96E7BE38 (used by other parts of the program) |
| 0x10007F5C (SP) | 0x0000023A |

2) When ELEMENT is first called (at address 0x000001AE), it overwrites the LR as before, but this is acceptable as the previous contents of the LR has already been saved. It executes and returns to call ELEMENT the second time.

|  |  |
| --- | --- |
| **Link Register** | **Program Counter** |
| 0x000001B2 | 0x000001B2 |

|  |  |
| --- | --- |
| **Stack Address** | **Value** |
| 0x10007F60 | 0x96E7BE38 (used by other parts of the program) |
| 0x10007F5C (SP) | 0x0000023A |

3) When ELEMENT is called the second time (at address 0x000001B2), the same thing occurs as with the first call. However, after the execution returns from ELEMENT, the value at the top of the stack is popped and placed back into the LR.

|  |  |
| --- | --- |
| **Link Register** | **Program Counter** |
| 0x0000023A | 0x000001BA |

|  |  |
| --- | --- |
| **Stack Address** | **Value** |
| 0x10007F60 (SP) | 0x96E7BE38 (used by other parts of the program) |
| 0x10007F5C | ~~0x0000023A~~ |

4) When distrBF(…) tries to return to the C program (at address 0x000001BA), it can do so because the LR now correctly points to the next instruction in the main program.

**Is there any drawback of using the ELEMENT subroutine to perform the required task in the distrBF function? If so, what are they?**

Using the ELEMENT subroutine requires us to keep track of the starting addresses of the array, and the array indices to read. It also requires us to allocate an additional register for storing the return value (or alternatively we can use the stack). The program will have to manually calculate the offset each time a value is to be read. A simpler approach would be to make use of the auto-increment feature of ARMv7-M to traverse the arrays, which would eliminate the need to use an ELEMENTS subroutine (and eliminate the need to recalculate the offset at every call).

Using the ELEMENT subroutine also makes the program slightly bigger than if we were to take advantage of auto-increment.

Another drawback of the current implementation of ELEMENT is that it assumes each cell is 32-bits in length. We may have trouble if we are trying to read char arrays which uses 8-bit cells. This can be fixed by having another parameter to specify the length of the cells (not implemented here).

**What can you do if you have used up all the available general purpose registers and you need to store some more values during processing?**

When we have used up all the registers, and still require more, we can start by identifying registers that are not crucial to the current part of the program’s computation. We can free these registers up by pushing their contents into the stack. When we are done using the registers, we can then pop the original values back from the stack into the respective registers and continue operation.

For example, if we need 5 registers, and all our registers are occupied, but only R5 to R10 contains crucial data for the next computation, we can push R0 to R4 into the stack. We can then use R0 to R4 for the computation. When we are done, we should pop the values from the stack back into R0 to R4.

# General Program Structure Overview

The general operation of the distrBF(…) function can be explained using a higher-level C-equivalent code. The code is as follows:

|  |
| --- |
| distrBF(T, (int\*)n\_res, (int\*)dij, (int\*)Dj) {  int N = n\_res[0][0];  for (int t=0; t<T; t++) {  int minDi = 0x7FFFFFFF;  int j = 0;  for (int j=0; j<N; j++) {  int currDi = dij[t][j]+Dj[0];  if (currDi < minDi) {  minDi = dij[t][j]+Dj[0];  minJ = j+1;  }  }  n\_res[t][0] = minDi;  n\_res[t][1] = minJ;  }  } |

We begin by first extracting the value of **N**, the number of nodes, from the first index of the **n\_res** array.

We then have two nested loops which allows us to loop through every node **j** for each timestep **t**.

For each timestep **t**, we need to find the associated minimum cost from node **i** itself (**minimum** **Di**), and the respective node **j** which gives us that minimum cost. Therefore, we first assume a worst case cost of 0x7FFFFFFF, then loop through every node and find the **current** **Di** through each node **j** (that is, **current** **Di = dij + Dj**). Whenever we find a better cost path (a **current** **Di** that is lower than the **minimum** **Di**), we save it as the new minimum **Di** for this timestep **t**, as well as the **j** through which we obtained this **Di**.

After looping through every node **j** at the end of each timestep **t**, we save the **minimum** **Di** and its respective node **j** into the appropriate **n\_res** slots. The process repeats for the next timestep **t** until there is no more data to be processed.

# Assembly Program Overview

Now that we have a good understanding of what we need to do, we can begin explaining the assembly side of things.

The assembly implementation of distrBF(…) is shown below:

|  |
| --- |
| //…Directives omitted  **distrBF:**  PUSH {R5, R6, R7, R8, R9, R10, R11, R14}  LDR R4, [R1]  MOV R5, #0  **loopT:**  MOV R10, #0x7FFFFFFF  MOV R6, #0  **loopN:**  MOV R9, #0  MOV R7, R2  BL ELEMENT  ADD R9, R8  MOV R7, R3  BL ELEMENT  ADD R9, R8  CMP R9, R10  ITT LT  MOVLT R10, R9  MOVLT R11, R6  ADD R6, #1  CMP R6, R4  BLT loopN  ADD R11, #1  STR R10, [R1], #4  STR R11, [R1], #4  ADD R5, #1  CMP R5, R0  BLT loopT  POP {R5, R6, R7, R8, R9, R10, R11, R14}  BX LR  @ Subroutine ELEMENT  **ELEMENT:**  MUL R8, R4, R5  ADD R8, R6  LSL R8, #2  LDR R8, [R7, R8]  BX LR  NOP  .end |

The following table shows the register scheme of the program:

|  |  |
| --- | --- |
| Register | Purpose |
| R0 | T (passed from main program) |
| R1 | n\_res (passed from main program) |
| R2 | dij (passed from main program) |
| R3 | Dj (passed from main program) |
| R4 | N |
| R5 | **t** counter |
| R6 | **j** counter |
| R7 | Array address passed to ELEMENT |
| R8 | Return value from ELEMENT |
| R9 | Current Di |
| R10 | Best (Minimum) Di (for current t) |
| R11 | Best j (for current t) |

# Assembly Program Implementation Details

R0 to R3 contains 4 parameters passed from the main program. The contents of these registers will not be modified for the rest of the implementation.

## Preparing Registers for Use

We need to push the values of all the registers we’ll be overwriting (from *R4* to *R11*, and *R14*/LR) into the stack. This is to avoid overwriting any existing data used by the main program. This is done using the PUSH instruction:

|  |
| --- |
| PUSH {R5, R6, R7, R8, R9, R10, R11, R14} |

These values are popped back at the end of distrBF(…) in the same manner.

## Retrieving N from n\_res

Because the **n\_res** array will eventually be overwritten with the results, we need to retrieve the value of **N** from the first index of **n\_res** and store it in *R4*. Since *R4* is a register, and **n\_res** is a memory location pointed-to by R1, we need to use the LDR instruction:

|  |
| --- |
| LDR R4, [R1] |

## Setting Up the Nested-Loops

We need to prepare two nested loops for this implementation. The outer loop loops through timesteps **t**, and the inner loop traverses nodes **j**. Therefore, the outer loop will loop using R5 as its counter, and the inner loop will use R6.

|  |
| --- |
| MOV R5, #0  **loopT:**  //…  MOV R6, #0  **loopN:**  //…  ADD R6, #1  CMP R6, R4  BLT loopN  //…  ADD R5, #1  CMP R5, R0  BLT loopT |

Notice that MOV is used to initialize the looping variable **t** in R5. To perform looping, CMP is used for R5 (**t**) and R0 (**T**), and BLT is used to loop if **t** < **T**. The same principle applies for the inner loop.

## Finding the Minimum Di

For each timestep **t**, we need to find a new **minimum Di**. Therefore, we need to reset R10 to the maximum possible value. Whenever we enter the inner loop, we retrieve the respective **dij** and **Dj** and sum them up. We then compare that sum with the **minimum Di** for this timestep, and store into R10 and R11 it if it’s better (smaller).

|  |
| --- |
| //…  **loopT:**  MOV R10, #0x7FFFFFFF  //…  **loopN:**  //… Di is summed and stored in R9  CMP R9, R10  ITT LT  MOVLT R10, R9  MOVLT R11, R6  //…  ADD R11, #1  STR R10, [R1], #4  STR R11, [R1], #4  //… |

The looping instructions have been omitted for clarity. Notice that an IT block is used here. R9 (**current** **Di**) is compared with R10 (**minimum** **Di** for this timestep) and if R9 is better than R10 then both the sum (in R9) and the node index (in R6) is stored into R10 and R11 respectively.

## Obtaining the Current Di

To obtain the current **Di**, we make use of the ELEMENT subroutine to retrieve the index. We then sum them up in R9.

|  |
| --- |
| MOV R9, #0  MOV R7, R2  BL ELEMENT  ADD R9, R8  MOV R7, R3  BL ELEMENT  ADD R9, R8 |

In our implementation, we specify the value of **N** in R4, the indices **t** and **j** in R5 and R6 respectively, and the starting address of the array in R7. After calling the subroutine, the value of the cell will be loaded in R8.

## The ELEMENT Subroutine

The ELEMENT subroutine calculates the address and retrieves the value stored in a 2D array given the array starting address, the array indices, and the row length.

This implementation obtains row length **N** in R4, row index in R5, column index in R6, and array starting index in R7.

To solve this problem, we need to first look at how arrays are organized in memory. Let us look at a simple 3x3 array stored in row major order (as in ARM Cortex-M3):

|  |  |  |  |
| --- | --- | --- | --- |
| Row / Column | 0 | 1 | 2 |
| 0 | a | b | c |
| 1 | d | e | f |
| 2 | g | h | I |

Given a starting address **x**, the data is actually stored as shown:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Address | x | x+4 | x+8 | x+12 | x+16 | x+20 | x+24 | x+28 | x+32 |
| Row | 0 | | | 1 | | | 2 | | |
| Column | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 |
| Value | a | b | c | d | e | f | g | h | i |

As observed, a 2D array is actually stored as a contiguous sequence of words. From the above table, we can clearly see that:

1. Each row increments the address by **N** words.
2. Each column increments the address by 1 word.

Therefore, given a starting address **x**, with row **i**, and column **j**, and row length **N**, we can obtain the exact cell address using the formula:

Offset = (i\*N+j)\*4

Cell Address = x+Offset

The above formula is implemented as shown:

|  |
| --- |
| MUL R8, R4, R5  ADD R8, R6  LSL R8, #2  LDR R8, [R7, R8] |

In the above code, R8 is used to calculate the offset (the same register is also used for returning the cell value at the end to the calling function). First, the row is multiplied by **N** and stored as the offset. Next, the number of columns is added to the offset. Finally, the offset is shifted by the left by 2 places (multiplying it by 4). The value of the cell [R7, R8] (value stored in the address pointed by R7+R8) is loaded back to R8 which is will be used by the distrBF(…) function.

# Appendix

## Statement of Contributions

Student 1: (Ang Kah Min, Kelvin, A0111890)

Student 2: (Sitti Maryam Binte Rashid Ridza, A0112675)

A. Joint Work in algorithm development, programming and report writing (briefly list a few specific

aspects):

* Developing the overall program structure. (The program structure in C)
* Planning the register scheme.
* Writing the loop structures in ASM.
* Storing the results and returning to the main program.
* Writing the Lab Manual Questions section.
* Writing the General Program Overview section.

B. Student 1’s Work in algorithm development, programming and report writing (briefly list a few

specific aspects):

* Developing the algorithm to find the minimum.

C. Student 2’s Work in algorithm development, programming and report writing (briefly list a few

specific aspects):

* Developing the algorithm to implement ELEMENT subroutine.

We both agree that the statements above are truthful and accurate.

Signatures and Names:

Ang Kah Min, Kelvin Sitti Maryam Binte Rashid Ridza

(Student 1) (Student 2)