Project #1

Assembly Analysis of a C Function

Zak Walton

EGR 424 – Dr. Parikh

6/1/2016

# Introduction

This objective of this project was for us to become more familiar with the ARM assembly language as well as the ARM development tool suite. The function analyzed for this project was the selsort() function which sorts a list of integers in ascending order using the selection sort algorithm. The c function was compiled into assembly code using the -Os option in order to compile for size over speed so that there was less assembly code to analyze. Each line of assembly code was analyzed at a register level as well as within the context of the selsort() function.

# C Source Code

**Figure 1** contains the source code for the selsort() function:

|  |
| --- |
| 1 /\*  2 \* Sort a list of integers using the Selection Sort algorithm.  3 \* The input parameters are an array of integers (Data) and  4 \* the number (N) of elements in the array. The array is sorted  5 \* in-place in ascending order.  6 \*/  7  8 void selsort(int Data[], int N)  9 {  10 int start, i, minix, temp;  11  12 for (start=0; start < N-1; start++)  13 {  14 minix = start;  15 for (i=start+1; i < N; i++)  16 {  17 if (Data[i] < Data[minix])  18 {  19 minix=i;  20 }  21 }  22  23 if (minix != start)  24 {  25 temp = Data[minix];  26 Data[minix] = Data[start];  27 Data[start] = temp;  28 }  29 }  30 } |

**Figure 1: Source Code of selsort()**

The source was compiled using the following command:

CC -DPREFER\_SIZE\_OVER\_SPEED -Os -S selsort.c

These options made sure to tell the arm gcc compiler to compile for size over speed as well as to generate an assembler output.

The way the code works is it starts at the beginning of the array of integers and then loops through the remainder of the array to find the smallest element left in the array after the current index. It then uses a temporary variable to swap the smallest value with the value in the current index if necessary, and then the index is incremented by one. This process repeats until the end of the array is reached and the whole list is sorted in ascending order. This function has a big O of O(n2) because of the nested loop required.

# Assembly Code

**Figure 2** shows the assembly code file that was trimmed down to only contain the assembly instructions and branch labels for the assembly subroutines. Each line in this file will be analyzed in depth.

|  |
| --- |
| 1 selsort:  2 push {r4, r5, r6, r7, r8, lr}  3 movs r2, #0  4 adds r4, r0, #4  5 b .L2  6 .L7:  7 adds r5, r2, #1  8 mov r6, r4  9 mov r3, r2  10 mov ip, r5  11 b .L3  12 .L5:  13 ldr r8, [r6], #4  14 ldr r7, [r0, r3, lsl #2]  15 cmp r8, r7  16 it lt  17 movlt r3, ip  18 add ip, ip, #1  19 .L3:  20 cmp ip, r1  21 blt .L5  22 cmp r3, r2  23 beq .L6  24 ldr r2, [r0, r3, lsl #2]  25 ldr r6, [r4, #-4]  26 str r6, [r0, r3, lsl #2]  27 str r2, [r4, #-4]  28 .L6:  29 adds r4, r4, #4  30 mov r2, r5  31 .L2:  32 subs r3, r1, #1  33 cmp r2, r3  34 blt .L7  35 pop {r4, r5, r6, r7, r8, pc} |

**Figure 2: Assebly Code for selsort()**

# Line-by-Line Analysis

Instruction #1, Line #2:

push {r4, r5, r6, r7, r8, lr}

Description/Big Picture:

This is the first instruction that is executed when the function is called, and its purpose is to save the state of r4-r8 on the stack so that they can be used throughout the function and restored before it returns. The lr is also stored on the stack so that the function knows where to return to when it exits.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | R2 | R2 |
| R3 | R3 | R3 |
| R4 | R4 | R4 |
| R5 | R5 | R5 |
| R6 | R6 | R6 |
| R7 | R7 | R7 |
| R8 | R8 | R8 |
| Sp(r13) | Sp | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort | Selsort + 4 |
| **APSR bit** | **Value** | **Rationale** |
| N | - | Unknown on reset |
| Z | - | Unknown on reset |
| V | - | Unknown on reset |
| C | - | Unknown on reset |

Instruction #2, Line #3:

movs r2, #0

Description/Big Picture:

This instruction moves a value of 0 into register r2. Register r2 is essentially used to store the value of the outside loop variable *start* throughout the function, so it makes sense that r2 is set to 0 in the beginning. Since the instruction has an s on the end it also updates the flags in APSR

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | R2 | 0 |
| R3 | R3 | R3 |
| R4 | R4 | R4 |
| R5 | R5 | R5 |
| R6 | R6 | R6 |
| R7 | R7 | R7 |
| R8 | R8 | R8 |
| Sp(r13) | Sp - 24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 4 | Selsort + 6 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0 | 0 is not negative |
| Z | 1 | Result is 0 |
| V | - | Unaffected |
| C | - | Untouched because no shift |

Instruction #3, Line #4:

adds r4, r0, #4

Description/Big Picture:

This instruction adds the immediate value of 4 to r0 and stores this into r4, and it also updates the flags in the APSR. This is because r4 is where the inner loop variable *i* is stored, and for each iteration through the outer loop the inner loop variable starts at the outer loop variable, *start,* plus 4. The reason it is 4 is because each element in the Data array is 32 bits wide so to go to the next element the pointer must be incremented by 4 bytes.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | 0 | 0 |
| R3 | R3 | R3 |
| R4 | R4 | R0 + 4 |
| R5 | R5 | R5 |
| R6 | R6 | R6 |
| R7 | R7 | R7 |
| R8 | R8 | R8 |
| Sp(r13) | Sp - 24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 6 | Selsort + 8 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0 | 0 is not negative |
| Z | 0 | Result is non-0 |
| V | 0 | No overflow |
| C | 0 | No Carry |

Instruction #4, Line #5:

b .L2

Description/Big Picture:

This instruction executes a branch to label .L2, which essentially just sets the program counter to the first instruction of the label subroutine. The .L2 subroutine is used to check if the condition to terminate the outside loop is satisfied, and either pops and returns if it is or continues into the loop if it isn’t.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | 0 | 0 |
| R3 | R3 | R3 |
| R4 | R4 | R4 |
| R5 | R5 | R5 |
| R6 | R6 | R6 |
| R7 | R7 | R7 |
| R8 | R8 | R8 |
| Sp(r13) | Sp - 24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 8 | Selsort + 66 (.L2) |
| **APSR bit** | **Value** | **Rationale** |
| N | 0 | Flags not updated |
| Z | 0 | Flags not updated |
| V | 0 | Flags not updated |
| C | 0 | Flags not updated |

Instruction #5, Line #7:

adds r5, r2, #1

Description/Big Picture:

This instruction is the first in the .L7 subroutine. The instruction adds 1 to r2 and stores the result in r5, updating the flags in the APSR. The point of r5 in the function is to store where *i* starts at for each iteration of the outer loop. Since r2 is *start*, *I* always ends up starting at r2 + 1.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | Pointer to Data[0] | Pointer to Data[0] |
| R1 | N | N |
| R2 | R2 | R2 |
| R3 | N-1 | N-1 |
| R4 | R4 | R4 |
| R5 | R5 | R2 + 1 |
| R6 | R6 | R6 |
| R7 | R7 | R7 |
| R8 | R8 | R8 |
| Sp(r13) | Sp - 24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 10 | Selsort + 12 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0 | Result non-negative |
| Z | 0 | Result is non-0 |
| V | 0 | No overflow |
| C | 0 | No Carry |

Instruction #6, Line #8:

mov r6, r4

Description/Big Picture:

This instruction moves the value in r4 into r6. The purpose of this instruction is to put the starting value of *i* into r6 so that it can just be incremented during the inner loop, while retaining the starting value of *i* for the current outer loop iteration in of r4.

This instruction moves the

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | Pointer to Data[0] | Pointer to Data[0] |
| R1 | N | N |
| R2 | R2 | R2 |
| R3 | N-1 | N-1 |
| R4 | R4 | R4 |
| R5 | R2 + 1 | R2 + 1 |
| R6 | R6 | R4 |
| R7 | R7 | R7 |
| R8 | R8 | R8 |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 12 | Selsort + 14 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0 | Flags not updated |
| Z | 0 | Flags not updated |
| V | 0 | Flags not updated |
| C | 0 | Flags not updated |

Instruction #7, Line #9:

mov r3, r2

Description/Big Picture:

This instruction moves the value in r2 into r3. The purpose of this instruction is to initialize *minix* to *start*, where r3 is *minix* and r2 is *start*. R3 is used throughout the inner loop to track the smallest integer that the loop has gone through.

This instruction moves the

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | R2 | R2 |
| R3 | N-1 | R2 |
| R4 | R4 | R4 |
| R5 | R2 + 1 | R2 + 1 |
| R6 | R4 | R4 |
| R7 | R7 | R7 |
| R8 | R8 | R8 |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 14 | Selsort + 16 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0 | Flags not updated |
| Z | 0 | Flags not updated |
| V | 0 | Flags not updated |
| C | 0 | Flags not updated |

Instruction #8, Line #10:

mov ip, r5

Description/Big Picture:

This instruction moves the value in r5 into ip(r12). The purpose of this instruction is to initialize the scratch register, ip, to the initial value of *i*. This is because ip can then be incremented throughout the inner loop execution, and the starting value of *i* can be retained in r5.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | R2 | R2 |
| R3 | R2 | R2 |
| R4 | R4 | R4 |
| R5 | R2 + 1 | R2 + 1 |
| R6 | R4 | R4 |
| R7 | R7 | R7 |
| R8 | R8 | R8 |
| Ip(r12) | Ip | R5(r2 + 1) |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 16 | Selsort + 18 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0 | Flags not updated |
| Z | 0 | Flags not updated |
| V | 0 | Flags not updated |
| C | 0 | Flags not updated |

Instruction #9, Line #11:

b .L3

Description/Big Picture:

This instruction branches to the first instruction under label .L3.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | R2 | R2 |
| R3 | R2 | R2 |
| R4 | R4 | R4 |
| R5 | R2 + 1 | R2 + 1 |
| R6 | R4 | R4 |
| R7 | R7 | R7 |
| R8 | R8 | R8 |
| Ip(r12) | R5(r2 + 1) | R5(r2 + 1) |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 18 | Selsort + 38 (.L3) |
| **APSR bit** | **Value** | **Rationale** |
| N | 0 | Flags not updated |
| Z | 0 | Flags not updated |
| V | 0 | Flags not updated |
| C | 0 | Flags not updated |

Instruction #10, Line #13:

ldr r8, [r6], #4

Description/Big Picture:

This instruction loads the value at the address contained in r6, and stores it in r8. R6 is then post-indexed by 4. In this case r8 is storing the value of Data[i], and r6 is being used for the address of Data[i]. The reason that r6 is post-indexed by 4 is so that next time around the loop this exact same instruction can be called again to harvest the next value in the list.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | start | start |
| R3 | minix | minix |
| R4 | &Data[start+1] | &Data[start+1] |
| R5 | Start + 1 | Start + 1 |
| R6 | &Data[i] | &Data[i+1] |
| R7 | R7 | R7 |
| R8 | R8 | Data[i] |
| Ip(r12) | i | i |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 20 | Selsort + 24 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0 | Flags not updated |
| Z | 0 | Flags not updated |
| V | 0 | Flags not updated |
| C | 0 | Flags not updated |

Instruction #11, Line #14:

ldr r7, [r0, r3, lsl #2]

Description/Big Picture:

This instruction loads the value at the address of Data[0] plus the offset of minix \* 4 and stores it into r7. This instruction is loading Data[minix] to be used in the comparison on line 17 of the c code.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | start | start |
| R3 | minix | minix |
| R4 | &Data[start+1] | &Data[start+1] |
| R5 | Start + 1 | Start + 1 |
| R6 | &Data[i+1] | &Data[i+1] |
| R7 | R7 | Data[minix] |
| R8 | Data[i] | Data[i] |
| Ip(r12) | i | i |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 24 | Selsort + 28 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0 | Flags not updated |
| Z | 0 | Flags not updated |
| V | 0 | Flags not updated |
| C | 0 | Flags not updated |

Instruction #12, Line #15:

cmp r8, r7

Description/Big Picture:

This instruction essentially does the subtraction of r8 – r7 and discards the result but updates the condition flags. This is used to execute the logic in line 17 of the c code.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | start | start |
| R3 | minix | minix |
| R4 | &Data[start+1] | &Data[start+1] |
| R5 | Start + 1 | Start + 1 |
| R6 | &Data[i+1] | &Data[i+1] |
| R7 | Data[minix] | Data[minix] |
| R8 | Data[i] | Data[i] |
| Ip(r12) | i | i |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 28 | Selsort + 30 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0/1 | 1 if r8 < r7, 0 if r8>r7 |
| Z | 0/1 | 1 if r8 = r7, 0 if r8!=r7 |
| V | 0/1 | 0 if number are sufficiently close, otherwise overflow if r7 >> r8 |
| C | 0/1 | Should be 0 if number are sufficiently close, otherwise carry if r8 >> r7 and r7 is negative |

Instruction #13, Line #16:

it lt

Description/Big Picture:

This instruction initiates an it block that contains 1 conditional instruction that should have ‘lt’ appended to it. This it block is used to conditionally execute the logic from line 19 of the c code based on the result of the compare on the previous line.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | start | start |
| R3 | minix | minix |
| R4 | &Data[start+1] | &Data[start+1] |
| R5 | Start + 1 | Start + 1 |
| R6 | &Data[i+1] | &Data[i+1] |
| R7 | Data[minix] | Data[minix] |
| R8 | Data[i] | Data[i] |
| Ip(r12) | i | i |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 30 | Selsort + 32 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0/1 | Flags not updated |
| Z | 0/1 | Flags not updated |
| V | 0/1 | Flags not updated |
| C | 0/1 | Flags not updated |

Instruction #14, Line #17:

movlt r3, ip

Description/Big Picture:

This instruction is conditional and will only execute if the result of the compare on line 15 is less than. It is the one instruction inside of the IT block. This instruction executes line 19 of the c code that sets *minix* to *i* which indicates it is the index of the smallest integer come across in this iteration of the inner loop so far.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | start | start |
| R3 | minix | i (which is the new minix) |
| R4 | &Data[start+1] | &Data[start+1] |
| R5 | Start + 1 | Start + 1 |
| R6 | &Data[i+1] | &Data[i+1] |
| R7 | Data[minix] | Data[minix] |
| R8 | Data[i] | Data[i] |
| Ip(r12) | i | i |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 32 | Selsort + 34 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0/1 | Flags not updated |
| Z | 0/1 | Flags not updated |
| V | 0/1 | Flags not updated |
| C | 0/1 | Flags not updated |

Instruction #15, Line #18:

add ip, ip, #1

Description/Big Picture:

This instruction adds one to the current value of the ip register, and stores the result in the ip register. This is equivalent to *i++* in the code.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | start | start |
| R3 | minix | minix |
| R4 | &Data[start+1] | &Data[start+1] |
| R5 | Start + 1 | Start + 1 |
| R6 | &Data[i+1] | &Data[i+1] |
| R7 | Data[minix] | Data[minix] |
| R8 | Data[i] | Data[i] |
| Ip(r12) | i | i + 1 |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 34 | Selsort + 38 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0/1 | Flags not updated |
| Z | 0/1 | Flags not updated |
| V | 0/1 | Flags not updated |
| C | 0/1 | Flags not updated |

Instruction #16, Line #20:

cmp ip, r1

Description/Big Picture:

This instruction subtracts the value in r1 from the value in ip, and updates the condition flags without storing the result of the subtraction. The purpose of this instruction is to check the inner for loop condition of *i < N* because ip is *i* and r1 is *N*.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | start | start |
| R3 | minix | minix |
| R4 | &Data[start+1] | &Data[start+1] |
| R5 | Start + 1 | Start + 1 |
| R6 | &Data[i+1] | &Data[i+1] |
| R7 | Data[minix] | Data[minix] |
| R8 | Data[i] | Data[i] |
| Ip(r12) | i | i |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 38 | Selsort + 40 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0/1 | 1 if ip < r1, 0 if ip>r1 |
| Z | 0/1 | 1 if ip = r1, 0 if ip!=r1 |
| V | 0/1 | 0 if number are sufficiently close, otherwise overflow if r1 >> ip |
| C | 0/1 | Should be 0 if number are sufficiently close, otherwise carry if ip >> r1 and r1 is negative |

Instruction #17, Line #21:

blt .L5

Description/Big Picture:

This instruction is a conditional branch that only branches if the condition in the APSR is less than. The purpose of this instruction is to enter another iteration of the inner for loop if *i < N*, otherwise continue. This condition was set up by the cmp on the previous line.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | start | start |
| R3 | minix | minix |
| R4 | &Data[start+1] | &Data[start+1] |
| R5 | Start + 1 | Start + 1 |
| R6 | &Data[i+1] | &Data[i+1] |
| R7 | Data[minix] | Data[minix] |
| R8 | Data[i] | Data[i] |
| Ip(r12) | i | i |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 40 | Selsort + 42 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0/1 | Flags not updated |
| Z | 0/1 | Flags not updated |
| V | 0/1 | Flags not updated |
| C | 0/1 | Flags not updated |

Instruction #18, Line #22:

cmp r3, r2

Description/Big Picture:

This instruction subtracts the value in r2 from the value in r3, and updates the condition flags without storing the result of the subtraction. The purpose if this instruction is to check the condition on line 23 of the c code. R3 represents *minix* and r2 represents *start*.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | start | start |
| R3 | minix | minix |
| R4 | &Data[start+1] | &Data[start+1] |
| R5 | Start + 1 | Start + 1 |
| R6 | &Data[i+1] | &Data[i+1] |
| R7 | Data[minix] | Data[minix] |
| R8 | Data[i] | Data[i] |
| Ip(r12) | i | i |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 42 | Selsort + 44 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0/1 | 1 if r3 < r2, 0 if r3>r2 |
| Z | 0/1 | 1 if r3 = r2, 0 if r3!=r2 |
| V | 0/1 | 0 if number are sufficiently close, otherwise overflow if r2 >> r3 |
| C | 0/1 | Should be 0 if number are sufficiently close, otherwise carry if r3 >> r2 and r2 is negative |

Instruction #19, Line #23:

beq .L6

Description/Big Picture:

This instruction is a conditional branch that only branches to .L6 if the condition in the APSR is equals. The purpose of this function is to start another iteration of the outside loop if minix != start, otherwise the data at start and minix must be swapped which is what happens in the next few lines if the branch doesn’t occur. This condition was set up by the cmp on the previous line.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | start | start |
| R3 | minix | minix |
| R4 | &Data[start+1] | &Data[start+1] |
| R5 | Start + 1 | Start + 1 |
| R6 | &Data[i+1] | &Data[i+1] |
| R7 | Data[minix] | Data[minix] |
| R8 | Data[i] | Data[i] |
| Ip(r12) | i | i |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 44 | Selsort + 46 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0/1 | Flags not updated |
| Z | 0/1 | Flags not updated |
| V | 0/1 | Flags not updated |
| C | 0/1 | Flags not updated |

Instruction #20, Line #24:

ldr r2, [r0, r3, lsl #2]

Description/Big Picture:

This instruction loads the data at (r0 + r3\*4) and stores it into r2. As far as the c function is concerned, the value at Data[minix] is stored into temp, which r2 is currently acting as. This executes line 25 in the c code.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | start | Data[minix] |
| R3 | minix | minix |
| R4 | &Data[start+1] | &Data[start+1] |
| R5 | Start + 1 | Start + 1 |
| R6 | &Data[i+1] | &Data[i+1] |
| R7 | Data[minix] | Data[minix] |
| R8 | Data[i] | Data[i] |
| Ip(r12) | i | i |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 46 | Selsort + 50 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0/1 | Flags not updated |
| Z | 0/1 | Flags not updated |
| V | 0/1 | Flags not updated |
| C | 0/1 | Flags not updated |

Instruction #21, Line #25:

ldr r6, [r4, #-4]

Description/Big Picture:

This instruction loads the data at &Data[start] and stores it into r6. As far as the c function is concerned, this instruction executes half of line 26 in the c code. The value at Data[start] is stored into r6.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | start | Data[minix] |
| R3 | minix | minix |
| R4 | &Data[start+1] | &Data[start+1] |
| R5 | Start + 1 | Start + 1 |
| R6 | &Data[i+1] | Data[start] |
| R7 | Data[minix] | Data[minix] |
| R8 | Data[i] | Data[i] |
| Ip(r12) | i | i |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 50 | Selsort + 54 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0/1 | Flags not updated |
| Z | 0/1 | Flags not updated |
| V | 0/1 | Flags not updated |
| C | 0/1 | Flags not updated |

Instruction #22, Line #26:

str r6, [r0, r3, lsl #2]

Description/Big Picture:

This instruction takes the data stored in r6 and places it at the address (r0 + r3\*4). This is executing the second half of line 26 in the c code by storing Data[start] into Data[minix]. None of the registers are changed by this instruction except for the pc.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | Data[minix] | Data[minix] |
| R3 | minix | minix |
| R4 | &Data[start+1] | &Data[start+1] |
| R5 | Start + 1 | Start + 1 |
| R6 | Data[start] | Data[start] |
| R7 | Data[minix] | Data[minix] |
| R8 | Data[i] | Data[i] |
| Ip(r12) | i | i |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 54 | Selsort + 58 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0/1 | Flags not updated |
| Z | 0/1 | Flags not updated |
| V | 0/1 | Flags not updated |
| C | 0/1 | Flags not updated |

Instruction #23, Line #27:

str r2, [r4, #-4]

Description/Big Picture:

This instruction takes the data stored in r2 and places it at the address (r4 - 4). This is executing line 27 in the c code by storing the old value at Data[minix] which is currently in r2 into the location of Data[start] to complete the swap. The only register updated by this instruction is the pc.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | Data[minix] | Data[minix] |
| R3 | minix | minix |
| R4 | &Data[start+1] | &Data[start+1] |
| R5 | Start + 1 | Start + 1 |
| R6 | Data[start] | Data[start] |
| R7 | Data[minix] | Data[minix] |
| R8 | Data[i] | Data[i] |
| Ip(r12) | i | i |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 58 | Selsort + 62 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0/1 | Flags not updated |
| Z | 0/1 | Flags not updated |
| V | 0/1 | Flags not updated |
| C | 0/1 | Flags not updated |

Instruction #24, Line #29:

adds r4, r4, #4

Description/Big Picture:

This instruction adds 4 to register r4 and stores the result back into r4 while updating the flags in the APSR. The purpose of this instruction is to increment the value in r4 by 4 so that it is contains the address of Data[start + 1] again for the next iteration of the outer loop in which start is incremented.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | Data[minix] | Data[minix] |
| R3 | minix | minix |
| R4 | &Data[start+1] | &Data[start+2] |
| R5 | Start + 1 | Start + 1 |
| R6 | Data[start] | Data[start] |
| R7 | Data[minix] | Data[minix] |
| R8 | Data[i] | Data[i] |
| Ip(r12) | i | i |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 62 | Selsort + 64 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0 | Int + 4 cant be negative here |
| Z | 0 | Int + 4 cant be zero here |
| V | 0 | No signed overflow here |
| C | 0 | Pointer plus 4 can’t cause carry here |

Instruction #25, Line #30:

mov r2, r5

Description/Big Picture:

This instruction moves the value in r5 into r2. Since r5 currently contains start+1 and r2 will be used as start in the next instructions, this is essentially executing the start++ for the outer loop.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | Data[minix] | start (old start++) |
| R3 | minix | minix |
| R4 | &Data[start+1] | &Data[start+2] |
| R5 | Start + 1 | Start + 1 |
| R6 | Data[start] | Data[start] |
| R7 | Data[minix] | Data[minix] |
| R8 | Data[i] | Data[i] |
| Ip(r12) | i | i |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 64 | Selsort + 66 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0 | Flags not updated |
| Z | 0 | Flags not updated |
| V | 0 | Flags not updated |
| C | 0 | Flags not updated |

Instruction #26, Line #32:

subs r3, r1, #1

Description/Big Picture:

This instruction subtracts 1 from r1 and stores the result into r3 while updating the flags in the APSR. This is calculating N-1 and putting it into r3 to be used for the outer for loop condition on line 12 in the c code.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | start | start |
| R3 | R3 | N-1 |
| R4 | R4 | R4 |
| R5 | R5 | R5 |
| R6 | R6 | R6 |
| R7 | R7 | R7 |
| R8 | R8 | R8 |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 66 | Selsort + 68 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0/1 | 1 if r1 < 1, otherwise 0 |
| Z | 0/1 | 1 if r1=1, otherwise 0 |
| V | 0/1 | 1 if r1 = 0Xffffffff (signed), otherwise 0 |
| C | 0 | Not possible |

Instruction #27, Line #33:

cmp r2, r3

Description/Big Picture:

This instruction subtracts r3 from r2 and updates the flags in the APSR but discards the result. This instruction is used to compare *start* to *N-1* for the outer loop on line 12 in the c code.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | start | start |
| R3 | N-1 | N-1 |
| R4 | R4 | R4 |
| R5 | R5 | R5 |
| R6 | R6 | R6 |
| R7 | R7 | R7 |
| R8 | R8 | R8 |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 68 | Selsort + 70 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0/1 | 1 if r3 > r2, 0 if r3<r2 |
| Z | 0/1 | 1 if r3 != r2, 0 if r3=r2 |
| V | 0/1 | 0 if number are sufficiently close, otherwise overflow if r2 << r3 |
| C | 0 | Should be 0 if number are sufficiently close, otherwise carry if r3 << r2 and r3 is negative |

Instruction #28, Line #34:

blt .L7

Description/Big Picture:

This instruction is a conditional branch that only branches if the condition in the APSR is less than. The purpose of this instruction is to enter another iteration of the outer for loop if *start < N-1*, otherwise continue. This condition was set up by the cmp on the previous line.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | start | start |
| R3 | N-1 | N-1 |
| R4 | R4 | R4 |
| R5 | R5 | R5 |
| R6 | R6 | R6 |
| R7 | R7 | R7 |
| R8 | R8 | R8 |
| Sp(r13) | Sp -24 | Sp -24 |
| Lr(r14) | Lr | Lr |
| Pc(r15) | Selsort + 70 | Selsort + 72 |
| **APSR bit** | **Value** | **Rationale** |
| N | 0/1 | Flags not updated |
| Z | 0/1 | Flags not updated |
| V | 0/1 | Flags not updated |
| C | 0 | Flags not updated |

Instruction #29, Line #35:

pop {r4, r5, r6, r7, r8, pc}

Description/Big Picture:

This instruction is used to pop all original values of r4-r8 from before the function call, as well as to pop the value from the link register that was stored on the stack into the pc so that the function can jump back to where it was called from after it terminates.

Registers/Flags:

|  |  |  |
| --- | --- | --- |
| **Register** | **Before Instruction** | **After Instruction** |
| R0 | &Data[0] | &Data[0] |
| R1 | N | N |
| R2 | start | start |
| R3 | N-1 | N-1 |
| R4 | R4 | R4 pre-selsort |
| R5 | R5 | R5 pre-selsort |
| R6 | R6 | R6 pre-selsort |
| R7 | R7 | R7 pre-selsort |
| R8 | R8 | R8 pre-selsort |
| Sp(r13) | Sp -24 | Sp |
| Lr(r14) | Lr | Lr (never changed) |
| Pc(r15) | Selsort + 70 | Lr |
| **APSR bit** | **Value** | **Rationale** |
| N | 0/1 | Flags not updated |
| Z | 0/1 | Flags not updated |
| V | 0/1 | Flags not updated |
| C | 0 | Flags not updated |

# Timing Analysis

**Table 1** below shows the cycle count of each instruction of the selsort() function.

**Table 1: Instruction Cycle Counts**

|  |  |  |
| --- | --- | --- |
| **Line** | **Instruction** | **Cycles** |
| **Selsort 1** |  | - |
| **2** | push {r4, r5, r6, r7, r8, lr} | 7 |
| **3** | movs r2, #0 | 1 |
| **4** | adds r4, r0, #4 | 1 |
| **5** | b .L2 | 1+P |
| **.L7 6** |  | - |
| **7** | adds r5, r2, #1 | 1 |
| **8** | mov r6, r4 | 1 |
| **9** | mov r3, r2 | 1 |
| **10** | mov ip, r5 | 1 |
| **11** | b .L3 | 1+P |
| **.L5 12** |  | - |
| **13** | ldr r8, [r6], #4 | 2 |
| **14** | ldr r7, [r0, r3, lsl #2] | 2 |
| **15** | cmp r8, r7 | 1 |
| **16** | it lt | 1 |
| **17** | movlt r3, ip | 1 |
| **18** | add ip, ip, #1 | 1 |
| **.L3 19** |  | - |
| **20** | cmp ip, r1 | 1 |
| **21** | blt .L5 | 1+P |
| **22** | cmp r3, r2 | 1 |
| **23** | beq .L6 | 1+P |
| **24** | ldr r2, [r0, r3, lsl #2] | 2 |
| **25** | ldr r6, [r4, #-4] | 2 |
| **26** | str r6, [r0, r3, lsl #2] | 2 |
| **27** | str r2, [r4, #-4] | 2 |
| **.L6 28** |  | - |
| **29** | adds r4, r4, #4 | 1 |
| **30** | mov r2, r5 | 1 |
| **.L2 31** |  | - |
| **32** | subs r3, r1, #1 | 1 |
| **33** | cmp r2, r3 | 1 |
| **34** | blt .L7 | 1+P |
| **35** | pop {r4, r5, r6, r7, r8, pc} | 7 |

It is was very difficult to perform a timing analysis on the selsort() function because the speed very much depends on the number of elements in the list as well as how out of order the list is. For this analysis, I assume worst case scenario where the list is in descending order to begin with so every element must be swapped. I have written equations for each subroutine below based on N and P, where N is the size of the list and P is the pipeline refill time.

Selsort:

(7+1+1+1+P) = 10+P

.L7:

(1+1+1+1+1+P) = 5+P

.L5 plus first two lines of .L3:

(n-1)\*((n-1)\*(2+2+1+1+1+1+1+1+P)) = (n-1)2\*(P+10)

Line 3-8 of .L3:

(n-1)\*(1+1+P+2+2+2+2) = (n-1)\*(P+10)

.L6:

2\*(n-1)

.L2:

(n-1)\*(1+1+1+P)+7

If all of the different subroutines are added together, this leads to the following equation:

T = (N2P + 10N2 - 5N + P + 17) cycles