NUS ECE [T]EE2028 Programming for Computer Interfaces

Assignment 1 – Semester 1, AY2024/25

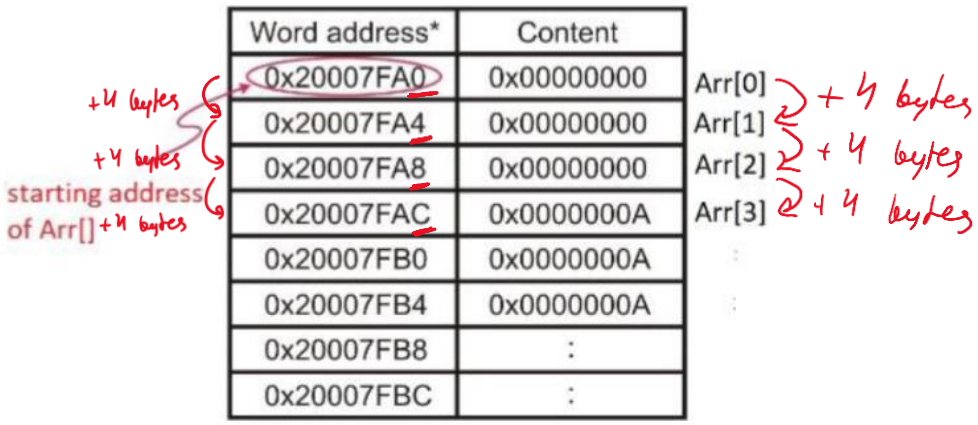
ARM v7-M Assembly Language and C Programming

Parv Bhadra

Anna Kavtaradze

Question 1:

When array is created, the address of the first value (which is also the smallest address in memory) is taken as a base address, and each following value is stored in the subsequent address in the memory (1word/4-bytes afterwards)



In case of Arr[], the address of Arr[0] is taken as base address. Arr[1] will be saved in the memory location directly after Arr[0]'s. in other words, it will be 4-bytes away from it's predecessor's address. So address of Arr[1] = address of Arr[0] + 4 bytes. Subsequently, we would need to increment by 4-bytes to reach the address of the next value in the array. To find address of Arr[A], we can, therefore, use this equation:

address of Arr[A] = address of Arr[0] + A * (4 bytes)

Question 2:

1.

   a. Before – the code went from main.c line 103 to optimize.s, and proceeded to execute it. it executed PUSH and updated SP to point to the memory location of the last data pushed into stack (R14 in this case, which is a special register LR, holding the memory location of the instruction that PC was pointing toward, had it not been manually overwritten to go to a different memory location. Essentially it holds the memory address of instruction in main.c we wish to return to after finishing all the instructions in optimize.s). Next, BL SUBROUTINE changes the contents of PC, resulting in changing the flow of the program and executing BX LR under SUBROUTINE before POP {R14} (since PC holds the memory address of the next instruction, if its contents change, it will point to a different memory location and therefore - a different instruction. In this case we change its contents so that it will point towards the memory location holding instructions under SUBROUTINE). Additionally, BL changes value of LR to hold memory address of instruction right after the BL (POP {R14} here). BX LR rewrites the contents of PC with whatever the LR is holding (in this case, address of instruction POP {R14}), returning the program flow back to optimize part. There, POP {R14} updates SP and brings the last value pushed into the stack (the one LR was initially holding and pointing back to the instruction in main.c) into R14(LR). Finally, BX LR returns the program back to main.c, since LR is now again holding the memory address of the instruction in main.c line 104.

   b. After – the program goes back into optimize, unlike the example above it does not save current value of LR in the stack, proceeding straight to instruction BL SUBROUTINE. Executing the command results in losing the initial value of LR, which was the only link that could point us back to the following instruction back in main.c (line 104 here). The program does not yet encounter any errors, so it continues to execute BX LR in SUBROUTINE, then (since there is no more POP {R14}) goes back to BX LR under SUBROUTINE and putting the contents of LR into PC. But LR is holding a value pointing to the instruction being currently executed, so the entire program freezes, without an option to step into, or step over the instructions.

2. One of the possible disadvantages of using POP and PUSH is that they are relatively expensive in terms of CPU cycles. Both of these operations need to access memory, which requires more time for the transfer and includes risks of the system being damaged or difunctional, leading to problems with transporting the data.

Question 3:

1. If the information some of the registers hold is not relevant anymore, we can simply rewrite them with new data
2. Create a label referring to a memory address and store content of a register in the memory address that label refers to
3. Store contents of several registers in memory right after one another and store memory address of only the first one in one of the registers. we would have to pay attention to the order in which

we stored all the contents, then afterward we would be able to retrieve any of them by using the base register saved in the register previously and offset.

4. Create a stack and push contents of different registers into the stack. we would also need to determine in advance which content we will need to retrieve first.

Question 4:

Link register would be pointing to printf() function call. Since foo is called in line 35, as discussed in Question 2 (since assembly code is identical to the one discussed in the question), at the highlighted point LR would hold address of the instruction in C code coming right after the one that called foo (the instruction that took us to the Assembly routine). Since foo is on line 35, the address in LR will be pointing to line 36 (the instruction right after 35).

Question 5:

1. It is meant to divide lowest between a and b (a in this case) by 2.0, so that we will retain the fraction from the division. If a is smaller than be, result will store the fraction obtained by dividing a by 2.0. otherwise (both options b<a and b=a) result will store fraction from division of b by 2.0
2. Depending on the data type we might not get the fraction from the division. In case of int, the fraction obtained from the division will be lost, and only the integer part will be stored in the result (the is the result from the division will be rounded down to the nearest integer). In the given example, since a<b (5<10), a would be the one divided by 2.0. Had result been an int type, it would have stored 2, whereas the float type would have the value of 2.5. consequentially, changing to int variable type would also mean changing to 'printf("Result: **%d**\n", result);'

Code:

1. Program flow:
   Once the code goes into optimiz.c, we save the contents of LR, so that we can return to the next instruction in main.c once the assembly part is done. Next we load values of variables a and b into the registers and multiply them by 2 and 10 respectively, since those are the values we will be continuously using in the loop. We also designate and set up the register to keep track of the number of rounds needed for the execution.
   Next we go into the loop and calculate what would be 'change' (in main.c this would be equivalent to '-1*change', since we do not multiply it by -1 in assembly code). since in the end, we are only interested in whether 'x' is equal to 'xprev', seeing if the difference between them (change) is 0 will suffice. Therefore, we only check if 'change' we calculated before is 0. Regardless of the answer we first update the number of rounds. Then if 'change' equals 0, we proceed to break off the loop, store 'x' (there is no need to update 'x', since the change would be

just adding 0) and the number of rounds elapsed in consecutive memory locations, then return pointer to those locations to the main.c code. Otherwise, we subtract the 'change' from 'x' to update its value (in main.c that would be similar to x = x – (-change)) and return to the beginning of the loop.

2. Improvements:

   The initial version of the code was a direct "translation" of the C rounded function to assembly.

   The obvious improvements are as follows:

   Instead of using separate registers for each variable as described in C rounded, registers are reused, limiting to R0-R6 and R8.

   Instead of branching in and out of the subroutine, a different method is used. When entering optimize, the value of R14(LR) is pushed to stack. After the loop is completed and answers are stored in memory, that value is directly popped to the program counter. This clarifies the initially convoluted program flow.

   Instead of checking if x has changed after adding the change, we check the value of change beforehand to overcome the unnecessary arithmetic operation in the last iteration.

Appendix:

Contributions:

Parv Bhadra– writing assembly code and Code.2: Improvements section of the report. 4-5x clock cycle count improvement for the code. Reviewing the report.

```
ASM version:
xsol : 0.2 No. of rounds : 3

asm() took 0.040000 seconds to execute
C version (accurate):
xsol : 0.1 No. of rounds : 14

C version (reference):
xsol : 0.2 No. of rounds : 3

C took 0.230000 seconds to execute
```

Anna Kavtaradze – writing all sections of the report except for code.2: Improvement section. Reviewing and editing the assembly code (suggested using .lcomm instead of manipulating and using address stored in R0 to store values needed in main.c code. Moved line ADD R6, #1 to be above BEQ exit).

joint – answering questions.