CG2028 Computer Organization Lab Assignment

Student 1: Liu Junhao A0230437M
Student 2: Ng Sihan, Ian A0231053X

Questions

Question 1: how to access the elements of an array created in main.c in the bubble_sort()? And, if the address of *N*-th element is X, what would be the address of the (*N*+*k*)-th element?

The elements of the array created in main.c can be accessed by first loading the starting address of the array into a register e.g. R0. R0 can then be used as a pointer to access the other elements in the array by adding the correct offset (each element is 4 bytes). If the address of the N-th element is X, then the address of the (N+k)-th element is X + 4k.

Question 2: Describe what you observed when the PUSH {R14} and POP {R14} instructions are not run. Is the change in the program's behaviour expected? Explain what happened.

The program was unable to return to the calling C program i.e. the main function. Assuming that we expect the program to return the sorted array and the total number of swaps, the program's behaviour is unexpected. This happened because we did not push the return address of the main function onto the stack before branching to the subroutine, and therefore after completing the subroutine, the program was unable to branch back to the main function since the memory address of the main function is not saved in the link register.

Question 3: Assuming that you have already pushed all the registers onto the stack. What else can you do if you have used up all the general purpose registers and you need to store some more values in the program?

One approach is to use the .lcomm assembler directive to create multiple variables (reserving a certain number of bytes in memory for each) which can then be used to store some more values in the program. An alternative approach is to branch to a subroutine and once again push the contents of the general purpose registers onto the stack, and then pop them in the correct order when finishing the subroutine.

However, this is against best practices as it is easy for the programmer to get confused about what the correct values of the registers should be at each juncture. The recommended best practice is that each register should only be pushed onto the stack once.

Program Logic and Algorithm

Bubble Sort is a simple sorting algorithm that works by repeatedly swapping the adjacent elements of an array if they are in the wrong order. With each linear pass of the array, the largest element of the subarray is “bubbled” towards the end of the array, hence giving the name Bubble Sort. In the event that no swaps are done in one of the passes over the array, the algorithm is finished and the program can be terminated as it is guaranteed that all elements are now in the correct order. This is known as optimised Bubble Sort.

Register	Purpose
R0	Used as a pointer to access the numbers in the array Register through which final answer (total number of swaps) is returned to the main function
R1	Start of program: Holds the value of M i.e. the number of integers in the array to be sorted As the program is executed: Holds the value of the size of the remaining subarray to be sorted (i.e. one past the last index of the subarray)
R2	Holds the number of swaps in each iteration (reset to 0 for every new iteration)
R3	Holds the total number of swaps done
R4	The index of the current number (the first number of the pair being compared)
R5	The index of the next number (the second number of the pair being compared)

R6	The current number (the first number of the pair being compared)
R7	The next number (the second number of the pair being compared)

As per best practices, upon first entering the assembly function, the contents of the registers used in the routine are pushed onto the stack. R14, or the Link Register, is pushed onto the stack as well so that we can return to our main program after the routine is complete. The value in R3 is also flushed and reset to 0 in preparation for the main sorting routine.

For every iteration/linear pass of the array (block of code with label: new), the values in R2 and R4 are set to 0 as we are starting from the leftmost position of the array each time.

During each iteration/linear pass of the array, we are considering each pair of numbers in the array in succession. This is done in the block of code with label: loop. We first add 1 to the index of the current number (R4) to get the index of the next number (R5). We then do a check to see if we have reached the end of the array (CMP R5, R1).

If we have not reached the end of the array, meaning to say R5 != R1, we will load the first (left) and second (right) number of the pair to be compared into R6 and R7 respectively. If R6 > R7, we will do a branch to the swapping routine (block of code with label: swap). Within this swapping routine, the value in R6 (bigger number) is stored at the memory address of the smaller number in the original array. Likewise, the value in R7 (smaller number) is stored at the memory address of the bigger number in the original array. Essentially, this means that we are swapping the numbers directly in the memory space of the array. We also increment the value in R2 by 1, which is currently tracking the number of swaps that have been done this iteration. Once the swapping routine is done, we will update the index of the current number to be that of the next number so that we can continue advancing rightwards in the array (MOV R4, R5). On the other hand, if R6 <= R7, no swap is needed so we will branch directly to the block of code with label: continue to move onto the next pair of the array.

The above loop will continue until we reach the end of the array i.e. R5 == R1. Now, we check if any swaps were done for the current iteration (CMP R2, #0) in an attempt to optimise the Bubble Sort. We first increment the total number of swaps (R3) by the number

of swaps done this iteration (R2). If at least one swap was done, the algorithm is not finished yet and we have to make another pass over the array. We then decrement the value in R1 by 1 (since the largest number has been propagated to the end of the array) and instantiate a new iteration by branching to the block of code with label: new. Otherwise, if no swaps were done, we continue on to the next instruction of moving the value in R3 (total number of swaps) into R0 so that the value can be returned to the main function. We then pop all the registers that we used off the stack and also pop the address in the Link Register into PC so that we can return to the main function (this effectively combines POP and BX LR into one instruction).

Possible enhancements to program

One possible modification to the program is to have R4 and R5 hold the actual numerical offset from R0 to decrease the complexity of the code as some programmers may not be fully acquainted with the operation of LSL. In addition, it might be possible to further refactor the code to replace the conditional branch instructions with IF-THEN blocks to increase the readability of the code.

Special efforts

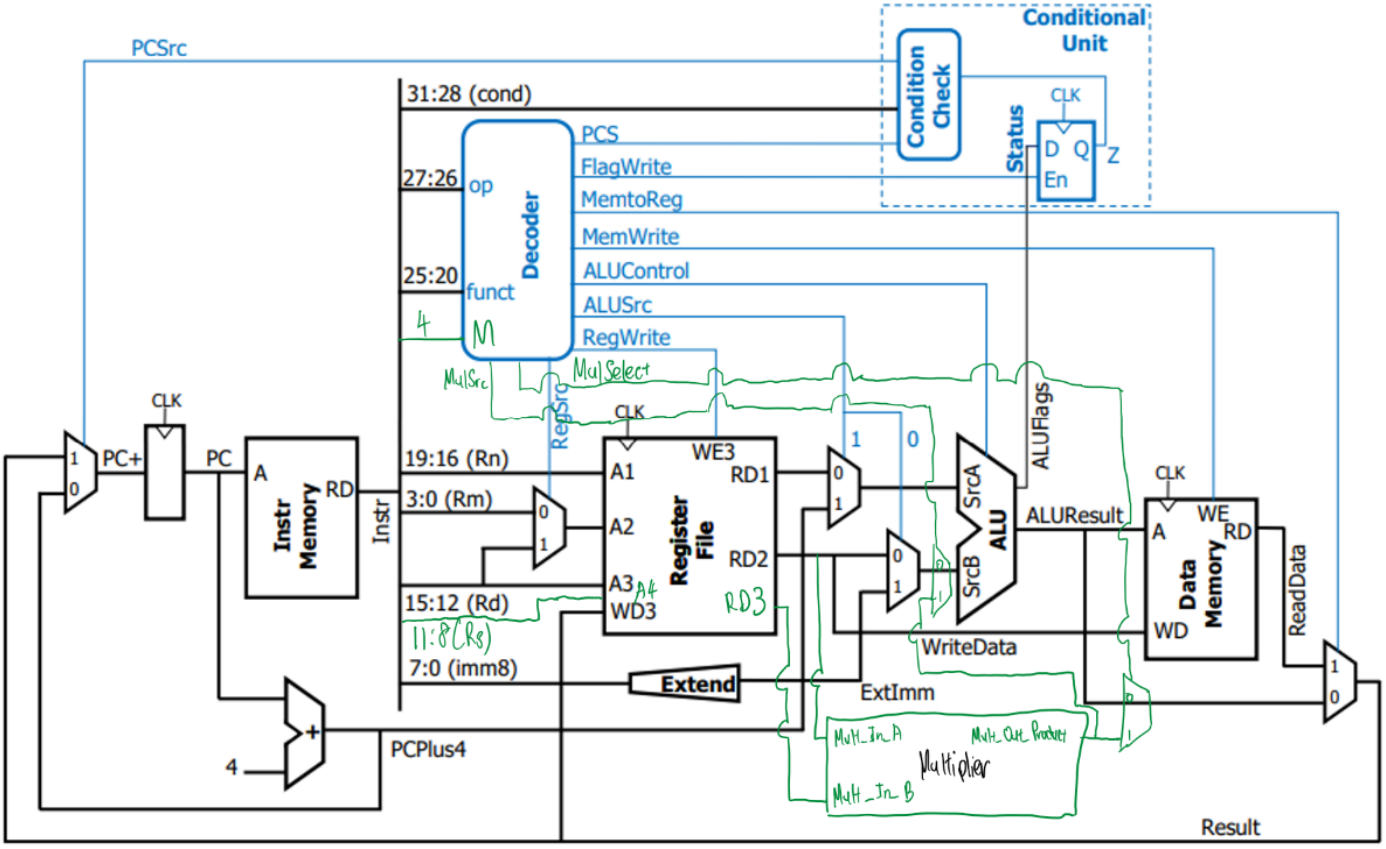
We have optimised the sorting routine of Bubble Sort to terminate when no swap is done for an iteration, avoiding the naive O(n^2) Bubble Sort solution. This way, we can return the correct answer earlier if possible and in the best case, this algorithm will run in O(n) time when the array is already sorted instead of bluntly comparing the pairs of numbers.

Annex

Source Code

```
bubble_sort:
        PUSH {R1-R7, R14}
        MOV R3, #0 @ machine code: 0x03A03000
new:
        MOV R2, #0
        MOV R4, #0
loop:
        ADD R5, R4, #1 @ machine code: 0x02845001
        CMP R5, R1 @ machine code: 0x01550001
        BEQ check @ machine code: 0x08800034
        LDR R6, [R0, R4, LSL #2]
        LDR R7, [R0, R5, LSL #2]
        CMP R6, R7
        IT GT
        BGT swap @ machine code: 0xC8800004
        B continue @ machine code: 0xE880000C
swap:
        STR R6, [R0, R5, LSL #2]
        STR R7, [R0, R4, LSL #2]
        ADD R2, #1
continue:
        MOV R4, R5
        B loop
next:
        SUB R1, #1 @ machine code: 0x02411001
        B new
check:
        ADD R3, R2 @ machine code: 0x00833002
        CMP R2, #0 @ machine code: 0x03420000
        BGT next @ machine code: 0xC8000014
end:
        MOV R0, R3
        POP {R1-R7, PC}
```

Paper Design for MUL and MLA instructions



The modifications made to the microarchitecture are denoted in green. For starters, the M bit which is the 4th bit in the instruction is fed into the Decoder. The MulSrc output bit from the Decoder is set when M is asserted ie. MulSrc = M. This signal is then passed into a multiplexer before SrcB of the ALU. If it is a multiplication instruction ie. the M bit is set, MulSrc is set and the multiplexer selects the Mult_Out_Product from the Multiplier block as SrcB into the ALU.

In addition, bits 11:8 of the instruction will also need to be considered as Rs. This is fed into the Register File as A4 and is delivered as the RD3 output. RD2 and RD3 serve as the Mult_In_A and Mult_In_B inputs to the Multiplier block respectively, to perform the Rm * Rs multiplication.

Now, depending on whether it is a MUL or MLA instruction, the MulSelect signal from the Decoder is necessary to distinguish between the two. The MulSelect output bit from the Decoder is set when it is just a plain MUL instruction ie. MulSelect = (cmd == 4b0000). This signal is then passed into another multiplexer that selects either ALUResult or Mult_Out_Product. If it is a plain MUL instruction ie. cmd = 4b0000, the multiplexer selects

Mult_Out_Product from the multiplier block. Otherwise, the MulSelect bit is cleared and the multiplexer selects ALUResult for all other values of cmd (which includes the MLA instruction with cmd bits corresponding to 4b0001).