# CG2028 Assignment 24/25 S2

Wong Weng Hong A0272156E
Tng Wen Xi A0282086Y

**Discussions of your program logic (overall logic flow, do not explain line by line)**

Our program logic is as such:

- Load F and S (from address stored in R3 and R3 + 4) into R4 and R5 and multiply them to get size of building array, load into R4
- Load totalSum of entryArr [R1] into R5
- Load loopsLeft (size of buildingArr, [R3] * [R3 + 4]), into R4
- Loop for each section in building
    - postIndex load currentSpace = [R0] (address of buildingArr)
    - Calculate availableSpace and insert car into currentSpace, then decrement totalSum
    - postIndex load currentExit = [R2]
    - Calculate remaining cars, currentSpace -= currentExit
    - postIndex store currentSpace into resultArr = [R3](address of resultArr)
    - Decrement loopsLeft (R4)
    - Exit loop when loopsLeft (stored in R4) < 1
- Branch Indirect to main.c

Our overall program logic can be further seen in Figure 1 below.

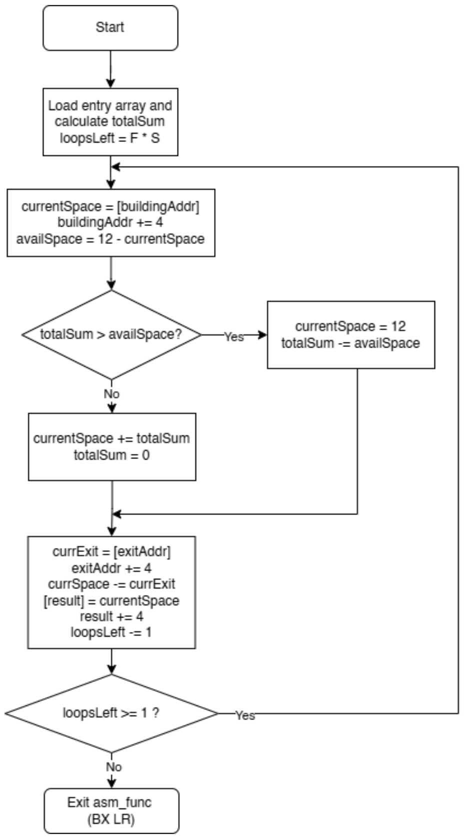
Start
Load entry array and calculate totalSum loopsLeft = F * S
currentSpace = [buildingAddr] buildingAddr += 4 availSpace = 12 - currentSpace
totalSum > availSpace?
Yes
currentSpace = 12 totalSum -= availSpace
No
currentSpace += totalSum totalSum = 0
currExit = [exitAddr] exitAddr += 4 currSpace -= currExit [result] = currentSpace result += 4 loopsLeft -= 1
loopsLeft >= 1 ?
Yes
No
Exit asm_func (BX LR)

**Figure 1: Overall Logic Flow**

**Question 1: Knowing the starting address of array `building[][]`, how to calculate the memory address of element `building[A][B]` with floor index `A` and section index `B`, with the index starting from 0? Use drawing or equation to explain your answer. (4 marks)**

Let the starting address of array Building[ ] [ ] be starting_address. The memory address of element building [A] [B] can be calculated by
starting_address + [ S * A + B ] * 4, where A, B are 0 - indexed.

When building is casted to int * as an argument to asm_func.s, it becomes a 1D array, which is stored as a contiguous chunk of memory. S * A calculates the total number of sections below the current floor and adding B gives the current section on the current floor. Multiplying by 4 as each section is stored as a word and each word is 4 bytes long.

**Question 2: Describe what you observe in (i) and (ii) and explain why there is a difference. (4 marks)**

In (i), the lines are commented, and the program is unable to return back to main.c.
In (ii), when the lines are uncommented, the program is able to return back to main.c and the expected results are printed.
When we push LR to the stack before branching with link to SUBROUTINE, the initial address of LR back to the instruction in main.c is saved in the stack, before LR is updated to the address of the instruction in asm_func. This allows LR to be restored by popping from the stack after the SUBROUTINE to return to main.c. If the initial address is not pushed to the stack, the address to the instruction in main.c is lost and the program will not be able to branch indirect back to main.c.

**Question 3: What can you do if you have used up all the general purpose registers and you need to store some more values during processing? (2 marks)**

The stack can be used by pushing registers onto the stack before performing operations. This ensures that their values are saved. Once the series of computations are completed, the stored values can then be popped back into the registers to restore their original values.

**5 machine codes correspond to 5 assembly language instructions in your code**

Line 48: LDR R4, [R3]
0000 0101 1001 0011 0100 0000 0000 0000
Line 49: LDR R5, [R3, #4]
0000 0101 1001 0011 0101 0000 0000 0100

Line 55: ADD R5, R6
0000 0000 1000 0101 0101 0000 0000 0110
Line 85: SUBS R4, #1
0000 0010 0101 0100 0100 0000 0000 0001

Line 88: BGT LOOP_COUNT
1100 1000 0000 0000 0000 0000 0011 1100

**Microarchitecture design that supports MLA and MUL instructions**

Our design implements the design with new signals, as shown in Figure 2 below.

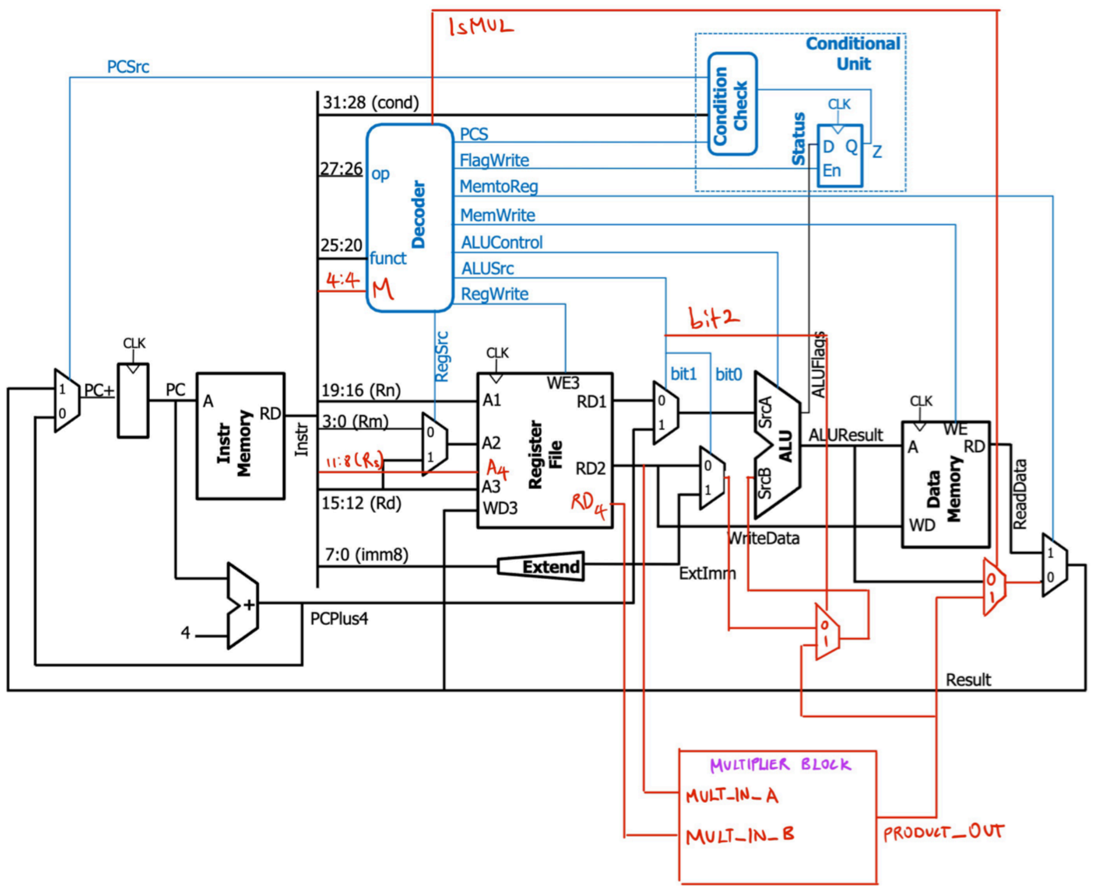

**Figure 2: Microarchitecture design that supports both MUL and MLA**

This design allows for normal operations as well as MUL and MLA, implemented with new signals in the decoder, ALUSrc[2] and IsMUL

ALUSrc[2] Checks if it is an MLA operation, if it is then addition in ALU is needed after the MULTIPLIER BLOCK, and SrcB of the ALU will take the value of the output of the MULTIPLIER BLOCK. It is defined as:

ALUSrc[2] = (op == 00) && (M = 1) && (I == 0) && (cmd == 0001) && (S == 0)

IsMUL checks if it is an MUL operation, and takes the result from the MULTIPLIER BLOCK if true, and else takes the result from ALUResult. It is defined as:

IsMUL = (op == 00) && (M = 1) && (I == 0) && (cmd == 0000) && (S == 0)

ALUControl is also changed to allow for the additional addition after multiplication for MLA, and it is defined as:

ALUControl = ( op == 00 ) ? ( M ? 0100 : cmd ) : ( U ? 0100 : 0010 )

This new microarchitecture design will support both MUL and MLA, including previous available operations in the lecture.

**Improvements you have made that enhance your program efficiency (reusing registers, more efficient algorithms, etc.).**

Improvements such as reusing registers, loop unrolling, minimizing memory access and more efficient algorithms have been used to enhance the program.

The registers have been reused, using only an additional 4 registers for our program (R4-R7), where R0-R3 had already been passed to the asm function by the C program. R1 is also reused in the middle of our program after we have read from the entry array. This allows us to use fewer registers, therefore, PUSHing and POPing fewer registers before and after the assembly program, improving the efficiency of the program.

Loop unrolling is used by repeating the code 5 times for reading the entry array instead of using a loop and branching, as the entry array is assumed to be always size of 5. This reduced the total number of operations in this section by more than half.

Only one loop has been used in our program. The building array, exit array and result array have been iterated at the same time, reducing the number of loops and branching performed, improving the efficiency of the program. At the same time, each of the addresses in the building and exit had only been loaded once while the address in the exit array had only been stored once, minimizing overall memory access as the number of load and store operations have been reduced, reducing the access overhead of memory operations.

The relevant comparison of snippets of code can be found in Appendix B.

## Appendix A

Appendix A declares every member's joint and specific individual contributions towards this assignment.

We did everything together in person, all the submitted files are of the joint efforts of Wong Weng Hong and Tng Wen Xi.

## Appendix B

Appendix B shows the before and after comparison of the code optimization.
Lookup Table Before Reusing Registers:

```
@ Look-up table for registers:
@ R0 - Building
@ R1 - Entry
@ R2 - Exit
@ R3 - Result
@ R4 - Size of building (number_of_loops_left index!)
@ R5 - Sum of entry cars
@ R6 - Size of entry (number_of_loops_left index!)
@ R7 - Temp register to load entry[A] and exit[A][B]
@ R8 - Size of entry (number_of_loops_left index!)
@ R9 - Temp register used in LOOP_COUNT to load building[A][B]
@ R10 - Store back to result[A][B]
@ R11 - Reused to store #12 for SUB operation, after LOOP_SUM
@ R12 - Temp register used to store available slots in each section
```

## Lookup Table After Reusing Registers:

```
@ Look-up table for registers:
@ R0 - Building
@ R1 - Entry and
@      Reused to store #12 for SUB operation, before LOOP_COUNT
@ R2 - Exit
@ R3 - Result
@ R4 - Size of building (number_of_loops_left index!)
@ R5 - Sum of entry cars
@ R6 - Size of entry (number_of_loops_left index!) and
@      Temp register used in LOOP_COUNT to load building[A][B]
@      and store back to result[A][B] and
@      Temp register used to store available slots in each section
@ R7 - Temp register to load entry[A] and exit[A][B]
```

## Before Loop Unrolling:

```
LOOP_SUM:
        LDR R7, [R1], #4 //ldr entry arr into R6, R5 is sum
        ADD R5, R7
        SUBS R6, #1
        BGT LOOP_SUM
```

## After Loop Unrolling:

```
        LDR R5, [R1], #4
        LDR R6, [R1], #4
        ADD R5, R6
        LDR R6, [R1], #4
        ADD R5, R6
        LDR R6, [R1], #4
        ADD R5, R6
        LDR R6, [R1], #4
        ADD R5, R6
```

## One loop to handle both car entry and exit:

```
LOOP_COUNT:
        LDR R6, [R0], #4 //store current section in buildingArr to R6
        LDR R7, [R2], #4 //store current section in exitArr to R7
        SUB R6, R1, R6 //R1 == 12
        //R6 is number of available slots in section
        CMP R6, R5
        ITT LE //if available slot <= total sum
                //if true then
                SUBLE R5, R6
                MOVLE R6, R1
        ITTT GT
                // else then
                SUBGT R6, R1, R6 //make R6 current number of used slots in building[a][b]
                ADDGT R6, R5
                MOVGT R5, #0

        //R6 is current section updated, proceed to exiting cars
        SUB R6, R7
        STR R6, [R3], #4

        SUBS R4, #1 // decrement number_of_loops_left counter,
                    // exit loop when R4 == 1 at before this line
                    // R4 == 0 after this line!
        BGT LOOP_COUNT
```