Program Logic

1. Load F and S from the result array. Compute $F \times S$.
2. Sum the 5 entry events from *R1* to find the total number of cars that entered, *C*.
3. For each section in the building array:
   a. Load the current building state from *R0*.
   b. Calculate the available capacity of each section (SECTION_MAX - building[i]).
   c. If *C* > 0, compare the available capacity to the waiting car count and either add all waiting cars if they fit, or fill the section to its maximum. Afterwards, update the waiting car total by subtracting the number of cars parked in the current section.
   d. Load the exit event (from *R2*) for the section and subtract its car count from the updated occupancy of the section.
   e. Store the final occupancy for the section into the results array.
4. Repeat the above steps for all sections before returning to the calling function in *main.c*.

Question 1

To access elements in an array, we would use offset addressing as elements of the array (regardless of 1D or 2D array) are stored in a contiguous block of memory, with the array pointer pointing to the address of the first element in the array.

The section values on each floor are stored consecutively in row-major order, where we can access the next section on the same floor by offsetting the base address we are starting from by 4 bytes as integer occupies 4 bytes. Once all the sections on the same floor have been accessed, the next 4-byte offset will store the value of the first section on the subsequent floor.

Therefore, the offset can be calculated with $((A * S) + B) * 4$, where A represents the floor index, B represents the section index, and S represents the total number of sections.

Question 2

The program does not terminate and exit from asm_func after commenting out the PUSH and POP commands, while it runs normally with PUSH and POP uncommented.

This is because R14 is the link register (LR), which stores the return address to the caller function (in this case, SUBROUTINE, and back to main.c). PUSH and POP pushes and pops the LR to and from the stack respectively.

In the program (with PUSH & POP), PUSH saves the caller address to return to the main program to the top of the stack. BL SUBROUTINE then branches to SUBROUTINE and saves the address of the next instruction in the LR, which happens to be POP. After returning from the SUBROUTINE with BX LR to the POP instruction, we POP the previously pushed value onto the

stack back into R14, which would be the return address from the asm_func to main.c. The following BX LR instruction then returns to asm_func from the caller function.

Without them, the original value to return from asm_func is not saved within memory as it gets overwritten in BL with the value of the next instruction, which happens to be BX LR. Hence, this creates an infinite loop within the asm_func and does not terminate as it would not have the correct address to return to.

Question 3

We can make use of the stack to temporarily store data or values. This is done by calling *PUSH* and *POP*, where *PUSH* stores the content of the register into the region of memory as per the stack pointer, and *POP* loads the content of the current stack pointer back into the register when we want to access the value again. This stack follows a LIFO architecture (Last In, First Out), where the last value that we push onto the stack gets popped out first.

We can also utilize direct memory storage if data exceeds the stack capacity, or needs to be stored beyond a function's lifetime. This is done through storing values at specific memory locations using STR instruction and retrieving them with LDR instruction. Specific memory areas can be allocated for storing immediate results or arrays, which require consecutive areas in memory.
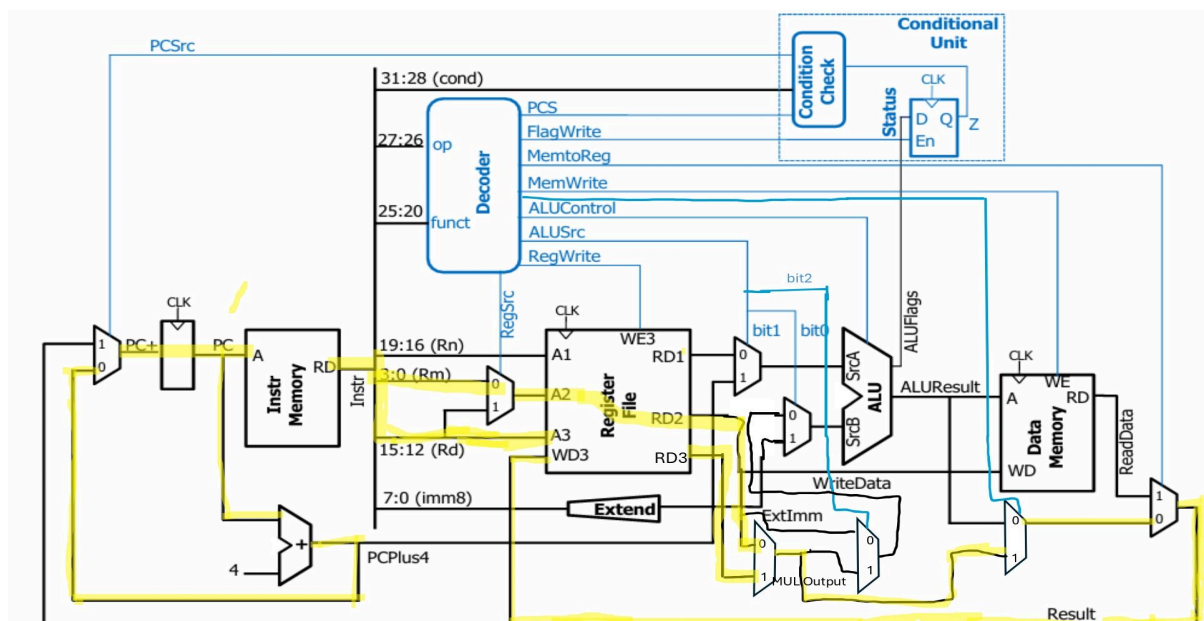
Machine Code

Assembly Code	Machine Code
(Memory Access) LDR R6, [R3]	0x05936000
(Data Processing) ADD R5, R5, R6	0x00855006
(Branching) BNE SUM_ENTRY_CARS_FOR_LOOP	0x18000010
LDR R8, [R3, #4]	0x05938004
MOV R5, #0	0x03A05000

Microarchitecture

MLA - Multiply & Accumulate
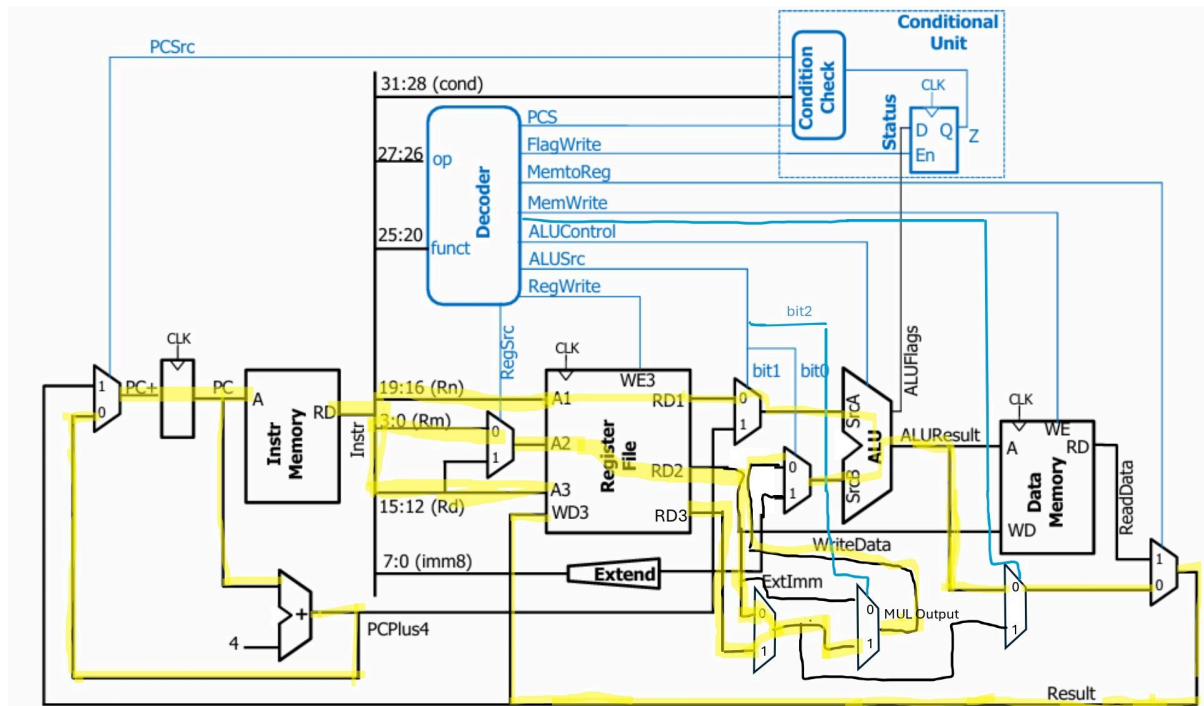
MUL - Multiply
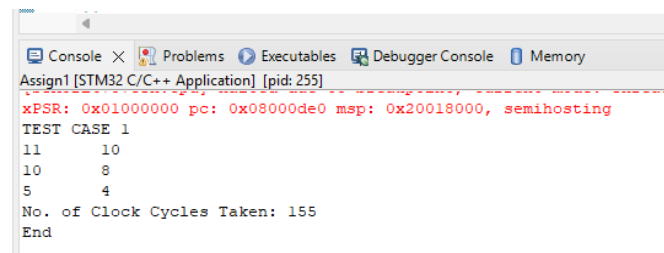
Improvements

Algorithm

We chose to sum all entry events in a single loop that sequentially adds each event to R5. Because of this, we can accumulate the total number of incoming cars just once, then perform the fill and subtract operations in a single pass for each section.

Our one-pass processing for the building array results in a consistent linear time complexity of $O(FS + E)$, where F is the number of floors, S is the number of sections and E is the number of entry events. If entry events were to be handled separately from exit events, the time complexity would still be $O(FS + E)$ in the best and average cases, but we would risk a worst-case time complexity of $O((FS)^2)$ as E increases and tends towards FS.

Implementation

Optimization for reduced clock cycles

The code has been optimized to significantly reduce the clock cycles required by employing several complementary techniques, resulting in about 150 clock cycles for most 3x2 test cases.

The effectiveness of these techniques were measured by measuring the clock cycles needed for the function in the *main.c* file, as shown in the screenshot in the *Appendix*.

Reducing the Number of Operations
The code minimizes its instruction count by combining operations wherever possible. For instance, using an instruction like SUBS R11, R11, #1 leverages the S flag to update condition flags simultaneously with subtraction, replacing what would otherwise require a separate CMP instruction (~20 clock cycles overall).

Moreover, we minimized the number of LDR and STR operations, PUSH/POP operations, as well as branching to further improve the throughput (~60 clock cycles overall).

Minimizing Instruction Overhead
Expensive operations like unnecessary PUSH/POP are avoided. Instead, post-index addressing (e.g., LDR R6, [R0], #4 and LDR R8, [R2], #4) is used to update pointers

automatically, eliminating the need for extra ADD instructions to manage indexing (~35 clock cycles).

Pipeline Optimization

Instructions are carefully reordered to hide load latencies. For example, after loading the building occupancy into R6, a comparison on R5 is executed immediately, allowing the processor's pipeline to remain busy while waiting for data dependencies to resolve. This reordering minimizes pipeline stalls and improves overall throughput (~10 clock cycles).

Opting for scalability

We chose to forgo some of the potential performance gains (~30 clock cycles) from unrolling our entry event processing. We felt that code readability and maintainability. Instead of hardcoding five LDR operations for a fixed number of entry events, we parameterized the count. This design decision makes the function more adaptable, allowing it to easily accommodate changes in the number of entry events in the future.

Appendix

Both members contributed equally to the project. Deng Jun did the basic version of the assembly code and did unit testing, while Yijian improved the code and optimized it. Both of us contributed equally towards the writing of this report.

Modified main.c to measure clock cycles

```
volatile unsigned int *DWT_CYCCNT   = (volatile unsigned int *) 0xE0001004;
volatile unsigned int *DWT_CONTROL  = (volatile unsigned int *) 0xE0001000;
volatile unsigned int *DWT_LAR      = (volatile unsigned int *) 0xE0001FB0;
volatile unsigned int *SCB_DHCSR    = (volatile unsigned int *) 0xE000EDF0;
volatile unsigned int *SCB_DEMCR    = (volatile unsigned int *) 0xE000EDFC;
volatile unsigned int *ITM_TER      = (volatile unsigned int *) 0xE0000E00;
volatile unsigned int *ITM_TCR      = (volatile unsigned int *) 0xE0000E80;

void initialise_timer() {
        *SCB_DEMCR |= 0x01000000;
        *DWT_LAR = 0xC5ACCE55; // enable access
        *DWT_CYCCNT = 0; // reset the counter
        *DWT_CONTROL |= 1 ; // enable the counter
}

int main(void) {
        initialise_timer();
        initialise_monitor_handles();

        int i,j;
        int building[F][S] = {{8,8},{8,8},{8,8}};
        int entry[5] =  {1,2,3,4,5};
        int exit[F][S] = {{1,2},{2,3},{3,4}};
        int result[F][S] = {{F,S},{0,0},{0,0}};

        int start_t = *DWT_CYCCNT;
        asm_func((int*)building, (int*)entry, (int*)exit, (int*)result);
        int end_t = *DWT_CYCCNT;
```

```
        // Prints final cycles taken (8 clock cycles OH)
        printf("# CLK Cycles Taken: %d\n", end_t - start_t);
        printf("End \n");
.....
```