1. Overview

This report details the design and implementation of an efficient assembly language solution to the engineering problem given. In short, it performs calculation to determine each value in the impulse response output sequence (y[1], y[2], y[3]...y[n]).

$$y[n] = b_0x[n] + b_1x[n-1] + ... + b_Nx[n-N] = \sum_{i=0}^{N} b_i \cdot x[n-i] \tag{1}$$

This report also seeks to explain how our assembly language program works, present the corresponding machine code as well as discuss the microarchitecture design incorporating both MUL and MLA instructions.

Note: All the machine code presented here follows the encoding format given in Lecture 4 and explanations are given for more complicated code.

2. Usage of Registers

When the assembly language function fir() is called, the C program passes 3 parameters into it through registers R0, R1 and R2. In addition, the return value of the assembly language function is passed back to the C program through register R0. Hence, we have:

R0: Stores N, which is is the order of the filter passed in by the C program, and returns y_n (or y[n]), which is the output signal to the calling C program.
R1: Stores the pointer to an array containing N + 1 filter coefficients (b[0] to b[N]) passed in by the C program.
R2: Stores the current x[n] i.e. x_n, which is the input signal passed in by the C program.

Furthermore, we make use of 6 additional general purpose registers for various tasks described below:

R3: Stores the calculated value of y_n (or y[n]).
R4: Stores either current or next value of filter coefficients in the array of b, i.e. b[j] or b[j + 1].
R5: Stores the memory address of STORE, a label declared to reserve space in the memory to store x_store values.
R6: Stores the required value of x_store (used as temp) loaded from the memory.
R7: Stores the constant decimal value of 10000.
R8: Stores N and is used as a counter for both the first and second loop.

3. ARMv7-M ASM Program

This section explains our implementation of the fir() function in assembly language. Our code can be categorised into four main parts as shown below.

3.1 Setup

This is the first part (out of four) of our assembly language function fir().

```
1 fir:
2         PUSH {R3-R8}
3
4         MOV R8, R0           @ 0000 00 011010 0000 1000 0000000 0 ↩
             0000 (0x01A08000)
5         LDR R4, [R1], #4     @ 0000 01 001001 0001 0100 0000 ↩
             00000100 (0x04914004)
6         MUL R3, R2, R4       @ 0000 00 000000 0000 0011 0100 000 1 ↩
             0010 (0x00003412)
7         LDR R5, =STORE
```

Line 2: As we will use R3-R8 during the program, we push them at the beginning of the program to preserve the content stored in these registers.

Line 4: Moves N (an immediate constant passed from the C program) from R0 to R8.

Line 5: Loads R4 with the content of memory address referenced by R1, adds 4 (the offset value) to the memory address referenced by R1 (Offset Addressing with Post Index).

Line 6: Multiplies the content of R2 and R4 and stores the result in R3. This is equivalent to performing R3 = b[0] * x[n].

Line 7: Loads the memory address associated with the label STORE into R5 (Pseudo-Instruction).

3.2 First Loop: Calculating y_n (or y[n])

This is the second part (out of four) of our assembly language function fir().

```
1  loop:
2          LDR R4, [R1], #4     @ 0000 01 001001 0001 0100 0000 ↩
              00000100 (0x04914004)
3          LDR R6, [R5], #4     @ 0000 01 001001 0101 0110 0000 ↩
              00000100 (0x04956004)
4          MLA R3, R4, R6, R3   @ 0000 00 000010 0011 0011 0110 000 ↩
              10100  (0x00233614)
5
6          SUBS R8, #1          @ 0000 00 100101 1000 1000 0000 ↩
              00000001 (0x02588001)
7          BNE loop             @ 0001 10 000000 000000000000 00010100↩
               (0x18000014)
8          @ imm8 = PC+4-BTA = 0d20
9
10
11         MOV R8, R0           @ 0000 00 011010 0000 1000 0000000 0 ↩
              0000 (0x01A08000)
12         SUB R8, #1           @ 0000 00 100100 1000 1000 0000 ↩
              00000001 (0x02488001)
13         SUB R5, #4           @ 0000 00 100100 0101 0101 0000 ↩
              00000100 (0x02455004)
```

Line 2: Loads R4 with the content of memory address referenced by R1, adds 4 (the offset value) to the address referenced by R1 (Offset Addressing with Post Index).

Line 3: Loads R6 with the content of memory address referenced by R5, adds 4 (the offset value) to the address referenced by R5 (Offset Addressing with Post Index).

Line 4: Multiplies the content previously loaded into R4 and R6 (in Line 1 and Line 2), adds the result to R3 and stores the final result in R3. This is equivalent to performing y[n] = y[n] + b[j + 1] * x_store[j].

Line 6: Subtracts 1 from the value in R8 and update the condition flags. SUBS is used here as this step requires the flags to be set to use in the next line (Line 7). (counter for looping). The machine code has two consecutive 1000 because R8 is both the R_n and R_d.

Line 7: Checks if Z = 0. If so, return to line 2, else, proceed to line 10 (MOV R8, R0).

Line 11: Moves N (an immediate constant) from R0 to R8. This is to reload the register R8 which is used as a counter, with the value N.

Line 12: Subtracts 1 from the value in R8 as the store loop in the subsequent section only

needs to execute N - 1 times. SUB is used here as this step does not require the flags to be set.

Line 13: Decrements the address referenced by R5 by 4, so that it points back to the last value in the STORE "array". SUB is used here as this step does not require the flags to be set.

3.3 Second Loop: Storing Values of x_store

This is the third part (out of four) of our assembly language function fir().

```
1 store:
2         LDR R7, [R5, #-4]    @ 0000 01 010001 0101 0111 0000 ↩
             00000100 (0x05157004)
3         STR R7, [R5]         @ 0000 01 010000 0101 0111 0000 ↩
             00000000 (0x05057000)
4         SUB R5, #4           @ 0000 00 100100 0101 0101 0000 ↩
             00000100 (0x02455004)
5
6         SUBS R8, #1          @ 0000 00 100101 1000 1000 0000 ↩
             00000001 (0x02588001)
7         BNE store            @ 0001 10 000000 000000000000 00010100↩
              (0x18000014)
8         @ imm8 = PC+4-BTA = 0d20
```

Line 2: Loads R7 with content from the memory address referenced by R5 minus 4 (Offset Addressing with offset).

Line 3: Stores the value previously loaded into R7 (in Line 2) into the memory address referenced by R5 (Offset Addressing with no offset).

Line 4: Subtracts the memory address referenced by R5 by 4. SUB is used here as this step does not require the flags to be set.

Line 6: Subtracts 1 from the value in R8 and update the condition flags. SUBS is used here as this step requires the flags to be set to use in the next line (Line 7). (counter for looping)

Line 7: Checks if Z = 0. If so, return to line 2, else, proceed to the next part of the program (Completion of Program).

3.4 Completion of Program

This is the last part of our assembly language function fir().

```
1          STR R2, [R5]         @ 0000 01 010000 0101 0010 0000 ↩
              00000000 (0x05052000)
2          MOVW R7, #0x2710
3          UDIV R0, R3, R7
4
5          POP {R3-R8}
6          BX LR
7
8  .equ N_MAX, 10
9  .lcomm STORE 4 * N_MAX
```

Line 1: Stores the value in R2 to the memory address of R5. This is equivalent to performing x_[0] = x_n.

Line 2: Moves the decimal number 10000 (an immediate value) to R7. MOVW is used as it is able to move a 16-bit immediate value which is required in this case to move 10000 into the register R7.

Line 3: Performs an unsigned division of value in R3 by value in R7 and stores the result in R0. The result is stored in R0 so that the result can be returned to the calling C program.

Line 5: Since R3-R8 were previously pushed, they are popped back at the end of the program to restore the original contents in the registers.

Line 6: Returns to the calling C program.

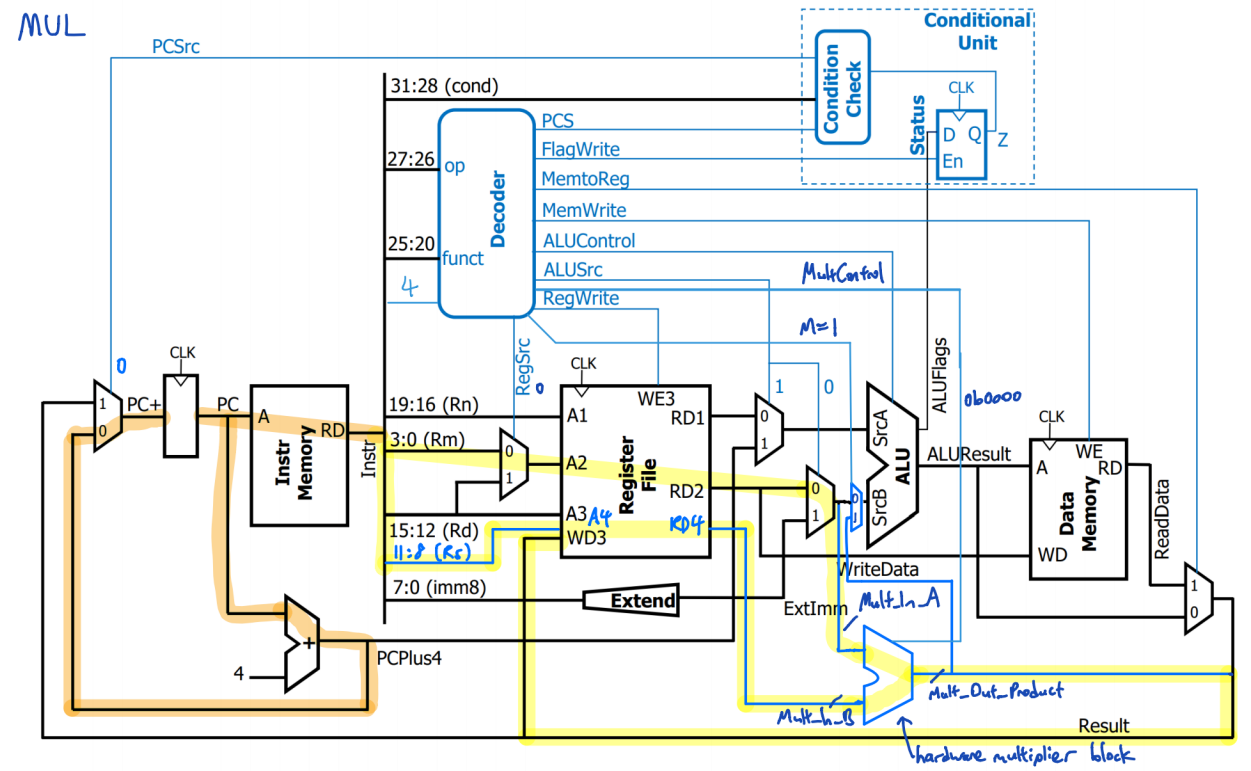
Lines 8 and 9: These are data declarations, by using assembler directive(s) at the end of an asm function. Specifically, .equ directive gives a symbolic name to a numeric constant, a register-relative value or a PC-relative value. Meanwhile, .lcomm defines a local uninitialised block of storage, in this case, 4 * N_MAX means that N_MAX spaces of 4 bytes wide are reserved for the storage of x_store values. Given below is a table to further illustrate this.

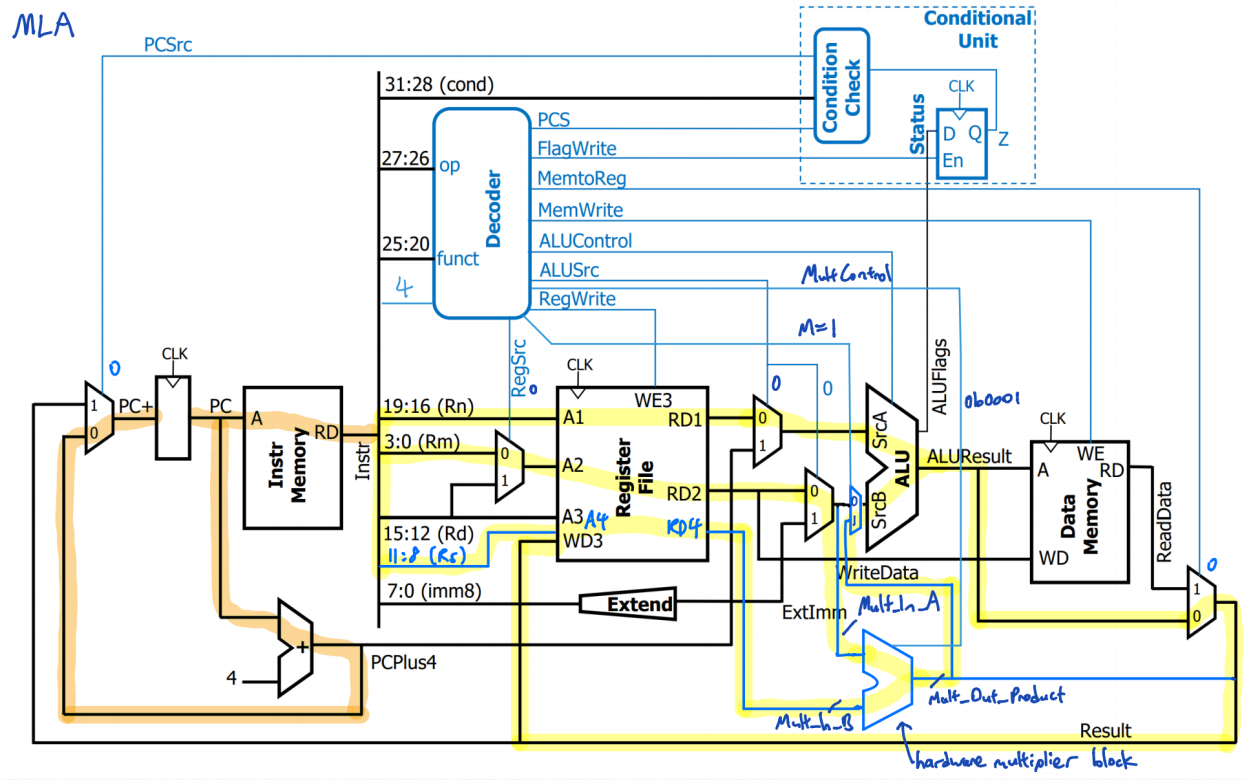
Memory Address (hex)	Label	Memory Contents
10	STORE	x_store[0]
14		x_store[1]
18		x_store[2]
1C		x_store[3]
20		x_store[4]
24		x_store[5]
28		x_store[6]
2C		x_store[7]
30		x_store[8]
34		x_store[9]

Note: The memory addresses given are not actual values. They are just arbitrary values used to illustrate the example.

4. Microarchitecture Design

Given below is the microarchitecture design, modified from the given microarchitecture in Lecture 4 (page 28) to support both MUL and MLA instructions.





Note: Both MUL and MLA diagrams have exactly the same hardware modifications. They differ in showing the datapath of the MUL and MLA operations respectively.

The modifications are hand-drawn in blue pen. All other parts remain exactly the same as the microachitecture given in Lecture 4 (page 28). The yellow coloured highlighter highlights the datapath of the MUL and MLA operations.

The decoder takes in an additional 1-bit input M (bit 4 of the instruction), which refers to whether multiplication operation exists. Specifically, the input M is 1 if multiplication instruction is required and is 0 if otherwise.

If op is 00 (which means it is a data processing instruction), I is 0 (which means there is no immediate value involved) and M is 1 (which means a multiplication operation is required), then it is conclusive for a multiplication instruction to be present. The decoder will then pass out MultControl signal, which is a 4-bit signal that contains cmd of MUL (0b0000) or MLA (0b0001) to the hardware multiplier block to perform either MUL or MLA. In other words:

$$MultControl is 0000 if ((op == 0b00) \&\& (I == 0) \&\& (M == 1) \&\& (cmd == 0b0000)). \quad (2)$$

$$MultControl is 0001 if ((op == 0b00) \&\& (I == 0) \&\& (M == 1) \&\& (cmd == 0b0001)). \quad (3)$$

In addition, we add another input register, R_s (bits 11:8 of instruction). Correspondingly, there is a new output RD4 which contains the memory content of R_s.

Lastly, to make sure the ALU is able to perform addition as part of the MLA instruction and continue to function as per normal for all other data processing instructions, we modify the original ALUControl (ALUControl = (op == 00) ? cmd : (U ? 0100 : 0010)) to the following:

$$ALUControl = (op == 2'b00)?((M\&\&!I)?4b'0100 : cmd) : (U?4'b0100 : 4'b0010) \quad (4)$$

This effectively allows the ALU to perform an ADD instruction for the two source operands in the MLA operation.