Assignment Details

1. Background Concepts

A function or subroutine can be programmed in assembly language and called from a C program. A well-written assembly language function could execute faster than its C counterpart.

In signal processing, a finite impulse response (**FIR**) filter is a filter whose impulse response (or response to any finite length input) is of finite duration because it settles to zero in finite time.

The impulse response (that is, the output in response to a Kronecker delta input) of an ***N*th** order discrete-time FIR filter lasts exactly ***N* + 1** samples (from first nonzero element through last nonzero element) before it settles to zero.

For an FIR filter of order ***N***, each value of the output sequence is a weighted sum of the most recent input values:

$$\begin{aligned} y[n] &= b_0 x[n] + b_1 x[n-1] + \cdots + b_N x[n-N] \\ &= \sum_{i=0}^{N} b_i \cdot x[n-i], \end{aligned}$$

where:

- ***x*[*n*]** is the input signal,
- ***x*[*n*-*i*]** is the input signal delayed by *i* samples,
- ***y*[*n*]** is the output signal,
- ***N*** is the filter order; an *N*th-order filter has (*N*+1) terms on the right-hand side
- b_i is the value of the impulse response at the *i*th instant for 0 *i N* of an *N*th order FIR filter, i.e., b_i is a coefficient of the filter.

Figure 1 will help you understand the operation of FIR filter better. Z^{-1} denotes one sample delay (the value on the right is a delayed version of the value on the left, i.e., it is the value on the left in the previous sampling period).

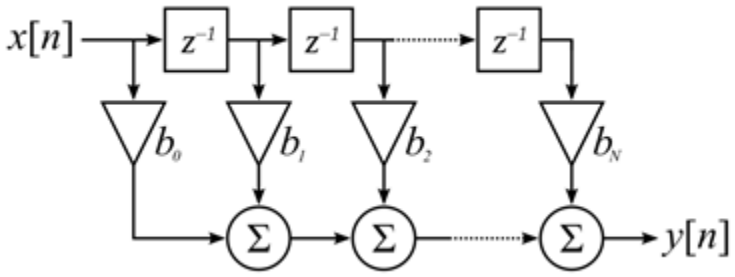

Figure 1 : A discrete-time FIR filter of order *N*

FIR filters could be high-pass, low-pass, band-pass or band-reject, depending on the coefficient values. FIR filters are widely used in processing sensor data as they are easy to design and implement, and has good phase characteristics.

2. Objectives

The objective of this assignment is to develop an ARMv7-M assembly language function which implements the function **int fir**(**int** N, **int*** b, **int** x_n);

where,

- **N** is the order of the filter. For memory allocation purposes, a constant N_MAX is defined (both in the main.c as well as fir.s files). The variable N has to be less than or equal to N_MAX.
- **b** is a pointer to an array containing N+1 filter coefficients (b_0 to b_N)
- **x_n** is the current *x*[*n*] passed to the function.

The return value of the **fir**() function is the current *y*[*n*].

The internal memory (for storing the delayed versions of x_n) is not guaranteed to be 0s when the main function starts executing. Hence the return values for the first N calls of **fir**() can be ignored. However, the function should return the correct *y*[*n*] from the N+1th call onwards.

You will also need to write down the machine code (8-digit hexadecimal, of the form 0xABCD1234) corresponding to each assembly language instruction as a comment next to the instruction in your assembly language function.

Note the following:

- You can assume that the instruction memory starts at a location 0x00000000.
- Follow the encoding format given in Lecture 4. The actual ARMv7M encoding format is different, but we will be sticking to the encoding format given in Lecture 4 for simplicity.
- Only the machine codes for the following instructions need to be provided.
    - Data processing instructions such as ADD, SUB, MOV[$], MUL, MLA, AND, ORR, CMP, etc., if they are used without shifts (i.e., the Operand2 is either a register without shift or imm8).
    - Load and Store instructions in offset, PC-relative, pre-indexed and post-indexed modes.
    - Branch instructions - conditional and unconditional, i.e., of the form B{cond} LABEL.

  [$]MOV is also one of the 16 DP instructions with the cmd 0b1101 as mentioned in slide 37 of Chapter 4. For MOV instruction, Rn is not used. You can encode Rn ($Instr_{19:16}$) = 0b0000. This makes sense as MOV has only one source operand which can be a register or immediate (recall: the assembly language format for MOV is MOV Rd, Rm or MOV Rd, #imm8), which means it can only come from the second source operand. Hence, the first source operand (which has to be a register, not immediate) is not used.

- You do not need to provide machine codes for instructions which do not fall into the categories above, even if you have used them in your program (e.g., BX, MOVW, multiplication instructions with 64-bit products such as UMULL/SMULL/UMLAL/SMLAL/SDIV/UDIV, division, data processing instructions where Operand2 is a register with shift etc.).
- Assume all instructions are 32 bits long, and that the assembler places the instructions and data (constants declared using .word) in successive word locations in the same order as they appear in assembly language. This should help compute the offset for PC-relative instructions, branches etc. Note that PC-relative mode is nothing but offset mode (PW=0b10) with Rn=R15.

You will also need to provide **a paper design showing how the microarchitecture** (datapath and control unit) covered in Lecture 4 can be modified to support MLA and MUL instructions. You can assume that a hardware multiplier block is available, which takes in two 32-bit inputs Mult_In_A and Mult_In_B, and provides a 32-bit output, Mult_Out_Product combinationally (i.e., without waiting for a clock edge).

3. Procedure

(a) Initial Configuration of Programs

The C program main.c in the project “CG2028AsmtS1AY202021” repeatedly calls **fir**() for different values of x_n and prints the corresponding y_n’s. Go through the C program carefully to understand its operation.

(b) Preparations

After completing the Self Familiarisation, you should have the “Lib_CMSISv1p30_LPC17xx” project and several other projects, in the Project Explorer pane of the LPCXpresso IDE (LXIDE).

The “Lib_CMSISv1p30_LPC17xx” project contains the Cortex Microcontroller Software Interface Standard (CMSIS) files.

Import the “CG2028AsmtS1AY202021.zip” archive file which contains the “CG2028AsmtS1AY202021” project. Within the “src” folder of this project, there are 3 files:

- *cr_startup_lpc17.c*, which is part of the CMSIS and does not need to be modified;
- *fir.s*, which presently does nothing and returns to the calling C program immediately - this is where you will write the assembly language instructions that implement the **fir**() function; and
- *main.c*, which is a C program that calls the fir() function with the appropriate parameters, and prints out the results on the console pane. You need not modify this file unless / until you wish to test your program with different values for the parameters.

In general, parameters can be passed between a C program and an assembly language function through the ARM Cortex-M3 registers. In the following example:

extern int fir(arg1, arg2, ….);

arg1 will be passed to the assembly language function fir() in the R0 register, arg2 will be passed in the R1 register, and so on. The return value of the assembly language function can be passed back to the C program through the R0 register.

(c) Procedure

Compile the “CG2028AsmtS1AY202021” project and execute the program.

Explain the console window output that you see.

Write the code for the assembly language function **fir**().

The assembly language program only needs to deal with integers.

You will need to allocate sufficient memory statically to store old values of x_n inside the assembly language function using .lcomm.

Verify the correctness of the results computed by the function you have written that appears at the console window of the LXIDE. A C language function **fir_c**()is provided as a reference. Note that it is fine for the first N calls of **fir**() to yield different y_n's as compared to **fir_c**(). However, their results should match from the N+1th call onwards.

Please note that the C function is provided just as a reference, and is NOT optimized. You are encouraged to implement a more optimized assembly language program.

There are several aspects that you need to pay attention to:

- It is a good practice to push the contents in the affected general-purpose registers onto the stack upon entry into the assembly language function, and to pop those values at the end of the assembly language function, just before returning to the calling main program.
- In a RISC processor such as the ARM, arithmetic and logical operations only operate on values in registers. However, there are only a limited number of general-purpose registers, and programs, e.g. complex mathematical functions, may have many variables.

Hint: Use and reuse the registers in a systematic way. Maintain a data dictionary or table to help you keep track of the storage of different variables in different registers at different times.

Show the assembly language program and actual console output to the Graduate Assistant (GA) during the assessment session.

Note: the assembly language program that you write should work for any value for any of the parameters. In other words, your program should work even if the values of *N_MAX*, *N* (not greater than N_MAX), *b* (which will have N+1 elements), *X_SIZE* (assume it is greater than N+1) are changed.

N_MAX will need to be changed separately in C (#define) and assembly (.equ) – there is no way you can pass *N_MAX* as a parameter, as you will be doing a static memory allocation in the assembly function to store old values of *x_n*.

4. Assessment

For each group of 2 students:

- Write a short report of about 4 pages long to explain how your assembly language program works, as well as the machine codes and microarchitecture design described in Section II above.
- For more information on the assessment schedule and process see the "Assessment and Submission info" tab

Some details about assessment:

- The assessing GA will modify your program slightly to see if your assembly language code is implemented correctly.
- The GA will verify the machine codes for each instruction (that you wrote as comment) during the assessment.
- You should also show and explain the paper design of your microarchitecture incorporating MUL and MLA instructions to the assessing GA. The design can be hand-drawn as long as it is legible – there is no need to spend your time on computer drawn schematics. Only the parts you have modified needs to be shown (redrawing the schematic already given in the lecture notes is not necessary).
- The Assignment mark for each student will consist of a group component and an individual component. The individual component will be based on a peer review as well as the perception of the assessing GA.
- The assessing GA(s) will ask each student some questions in turn during the assessment. Both students need to be familiar with all aspects of the assignment and your solution as the GA can choose any one of you to explain any part of the assignment. The GA will also ask you additional questions to test your understanding.
- To test your understanding of instruction encodings / machine codes, the GA could also ask you to figure out the encoding of any instruction whose format is specified in Lecture 4 even if you have not used that particular instruction in your program.

END