CE2007 Lab1 Assignment Sheet (to be submitted to NTULearn before next lab)

Name: Do Anh Tu Lab Group: SEP1 Date: 13/09/2019

1. Section 7.5. Why do we use SDIV instead of UDIV when calculating the Distance D?

SDIV is signed division. It is used as denominator can be <0 if ADC 14bit value n is <1058. This is totally possible since 14-bit value ranges from 0-2^14-1 in value. On the other hand, IR sensors have blind spot where the ADC value can be falling into 0 – 1058 ranges. Thus, this SDIV can let us know if we should ignore the value from the IR sensors to convert to distance.

1. Section 7.5. Why must the calling function save LR before calling another function?

Let us have 3 level of nested subroutine: Subroutine 1 calls subroutine 2, who calls subroutine 3.

When subroutine 1 calls subroutine 2, the return address to subroutine 1 (right after subroutine 2 is finished) will be saved into LR. Similarly, return address of subroutine 2 is saved into LR when it calls subroutine 3. However, this saving of subroutine 2’s return address overwrites the return address to subroutine 1 being saved earlier. Therefore, subroutine 2 cannot yields itself to return the program flow to the subroutine 1 (who is calling subroutine 2). In short, the overwrite in LR makes the subroutine 2 keep being called by itself again and again.

There is the need therefore to store the return address of the calling subroutine (1, 2, …) (on Stack most likely) before proceeds going into their nested subroutine, so that at the end of their called subroutine (2, 3, …), the system can pop this LR value and return to the upper level subroutine. This saving LR before calling another subroutine is to prevent the overwriting of data in LR whenever we have a few nested subroutines under each another.

1. Section 7.5. If the input parameters are passed in via R0-R3, how is it that the called function is allowed to freely modify these four registers?

As Arm processors follow a specification called AAPCS which enables transparent subroutine to be implemented easily (less worry for the programmers):

* Scratch registers (R0-R3) are used to pass parameters into subroutine. They are caller saved. Meaning before assigning parameters value to them and pass into the callee subroutine, the caller subroutine has already saved their context. As a result, the callee subroutine is free to change their value without worrying about saving and returning their context. When returning back to the caller subroutine, the caller subroutine will return the context to these registers to make sure transparent subroutine. Again, this is not the callee’s business. In ARM, this save and return of context for these regs are done automatically by hardware whenever a call to callee happen.
* Preserved Registers (R4-R11): Callee saved. Opposite to caller saved, the callee has responsible to save and return context of these registers if it was to use them in its body to ensure transparent subroutine rule.

1. Section 7.6. What does “ldr r1, [pc, #0x2e4]” do?

The assembly instruction loads the 32-bit value stored in address location of nextPC (which = current PC + 4) + #0x2e4 into register R1.

The LDR R1, [PC, #0x2e4] is stored in address 0xe4, hence the value loaded into R1 is from address location 0x3CC.

1. Section 7.6. In the code snippets shown, why is the same instruction “ldr r1, [pc, #0x2e4]” used in the initialization of Port1’s SEL0 and SEL1 registers? Does that mean these instructions are all writing to the same location?

The instructions are reused as after writing to the needed location in initialisation of SEL0, the register R0, R1 are free for other purposed and is hence reused.

No, these instructions do not write to same location (where addresses are stored in R1) as LDR instruction in initialization of SEL0 has PC = 0xe4, while the PC of LDR instruction to initialise SEL1 = 0xea instead.

As their PC are different, the resulted address location (= PC + 4 + 0x2e4) to be written into are different.

In short address stored in R1 in SEL0 and SEL1 initialisation are different due to different PC value at execution of the same LDR instruction. Hence, we are not writing to the same location

1. Section 7.6. Which register is used to store the return value of Port1\_Input()? Which register is used to store the argument of the Port1\_Output(data) function?

Register R0 is used to store return value of Port1\_Input() (conform to AAPCS).

Register R0 is also used to pass the argument data of Port1\_Output(data) function into the function subroutine.

This is based on AAPCS where register R0-R3 are used to pass the first four arguments of any function into it during subroutine call. And this standard also defines that R0 will be used to return the value need to be returned from the subroutine call when it is exited.

1. Section 7.7. How large is the code size for this project?

Code is allocated to .text software section. Thus from .map file, the size/length of the .text software section is 0x00000910 Bytes. Hence the code size of the project is 0x00000910 Bytes

1. Section 7.7. Which file consumes the largest code size in this project?

InputOutput.obj and system\_msp432p401r.obj both consumes 0x0000032c Bytes each and are of the largest code size in this project.

1. Section 7.7. How much SRAM is left for program expansion?

SRAM\_Code physical partition is left with 0x0000fed4 bytes and SRAM\_data physical partition is left with 0x0000fed4 bytes for program expansion. Assume both are used in expansion of the program.

1. Section 7.7. From the map file, what is the starting address of Port1\_Init()? Compare with the address you see in the Disassembly Window, are they the same? If not, why?

The starting address of Port1\_Init in the .map file and disassembly window are different. Starting address of Port1\_Init() in the .map file is 0x000000E5 but in the disassembly view is 0x000000E4.

Cortex-M4F supports Thumb-2 instruction set with Thumb (16-bit instructions) and ARM (32-bit instructions) states. Branch instructions does not give change in instruction set state, BX and BLX do changes the states of instruction set. BLX always give change to the other states but BX relies on the LSB of LR (bit[0]) to determine if there is a change in Thumb/Arm state to another or no.

If LSB = 0, processor changes to or remains in ARM state.

If LSB = 1, processor changes to or remains in Thumb state.

Therefore, a hard fault might occur or the incorrect state of the instruction set might be used. Hence all addresses are shifted by 1 bit to ensure all linking address are odd and therefore LSB of LR = 1 and so that the correct operation state of processor is achieved.