#cec32xNotes, cec32x_ws_30–WS06_Mixed_C_n_assembly_programming

WS 6. Mixed C and Assembly Programming

Introduction

In this project, we practice mixed C and assembly programming. Specifically, we program the 64-bit addition and subtraction functions using assembly instructions. These functions will be called in C.

For convenience, the main function in C and the skeleton of the assembly functions are given below. You can get started with the project unzipped from `expl_012_template_for_simulator_prjc_with_c_asm.zip` and replace the files there using the following. (You have to figure out how to do the replacement by using removing and adding.)

Full code in the `main_ws6.c` file:

```
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include <inttypes.h>

#include "ws6_add_sub_tc_asm_funcs.h"

uint64_t A[] = {0x87777777FFFFFFFF, 0x9876543200000000};
uint64_t B[] = {0x0000000300000002, 0x1111111122222222,
                0x0000000100000000};
uint64_t sum_A;
uint64_t sum_C;
uint64_t sub_A;
uint64_t sub_C;
int64_t tc_A;
int64_t tc_C;

int main(void) {
    determine_data_order(A[0], B[0]);

    sum_C = A[0] + B[0];
    printf("uint64 addition by C:   0x%" PRIX64 "\n", sum_C);
    sum_A = add_uint64_s(A[0], B[0]);
    printf("uint64 addition by asm: 0x%" PRIx64 "\n", sum_A);

    sub_C = A[1] - B[1];
```

```
    printf("uint64 subtraction by C:   0x%" PRIX64 "\n", sub_C);
    sub_A = sub_uint64_s(A[1], B[1]);
    printf("uint64 subtraction by asm: 0x%" PRIx64 "\n", sub_A);

    tc_C = - (int64_t) B[2];
    printf("TC of B[2] by C:   0x%" PRIX64 "\n", tc_C);
    tc_A = tc_uint64_to_int64_s(B[2]);
    printf("TC of B[2] by asm: 0x%" PRIx64 "\n", tc_A);

    while (1);
}
```

Full code in the `ws6_add_sub_tc_asm_funcs.h` file:

```
#ifndef __WS6_ADD_SUB_TC_ASM_FUNCS_H
#define __WS6_ADD_SUB_TC_ASM_FUNCS_H

#ifdef __cplusplus
 extern "C" {
#endif /* __cplusplus */

#include <stdint.h>

extern uint64_t determine_data_order(uint64_t x, uint64_t y);
extern uint64_t add_uint64_s(uint64_t x, uint64_t y);
extern uint64_t sub_uint64_s(uint64_t x, uint64_t y);
extern int64_t tc_uint64_to_int64_s(uint64_t x);

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* __WS6_ADD_SUB_TC_ASM_FUNCS_H */
```

Skeleton code `ws6_add_sub_tc_asm_funcs.s` file:

```
        AREA myfunctions, CODE

        EXPORT determine_data_order
        EXPORT add_uint64_s
        EXPORT sub_uint64_s
        EXPORT tc_uint64_to_int64_s
```

```
        ALIGN

determine_data_order  PROC

        BX     lr
        ENDP

add_uint64_s  PROC
        PUSH  {r4, r5}

        POP   {r4, r5}
        BX     lr
        ENDP

sub_uint64_s  PROC
        PUSH  {r4, r5}

        POP   {r4, r5}
        BX     lr
        ENDP

tc_uint64_to_int64_s  PROC
        PUSH  {r4, r5}

        POP   {r4, r5}
        BX     lr
        ENDP

        END
```

Note that we only use the simulator for this project. You may want to get started from an existing working project; then you just need to replace the `.c` and `.s` files using the given ones.

Programming Tasks

We have the following tasks:

Task 1—Checking the passing of the arguments from a C caller to an assembly function

(10 points)

We have learned that when passing arguments in 32 bits, `r0` to `r3` are used in the order of the arguments. Here, as shown in the C code, we are passing TWO 64-bit arguments, each of which takes TWO 32-bit registers, to the assembly function. Assume we do not know what registers are used to accept these agreements. Here we can call an empty function `determine_data_order` to determine how the registers are used. Note that we can see the passed variables in the registers by setting a breakpoint in the line of `determine_data_order` in the C program and step in the assembly function, which is **empty**. Upon stepping into the assembly function, we can see the values of `r0` to `r3`, from which we should be able to figure out how the agreements are passed to the function from their special values.

Artifacts will be needed in the submission of the report, as shown later.

Task 2—Programming an `uint64_t` addition function in assembly

(30 points)

We have learned how to do the 64-bit addition in the class using assembly instructions. In this task, we repeat the same work with the values passed by the caller. Use the code example in the class notes and the order you have just determined in Task 1 to finish this code. Note that you can use the registers `r0` to `r5` in this code—just make sure you cannot erase the data saved in `r4` and `r5` for the caller. This can be done by pushing and popping the values of `r4` and `r5`, as shown in the lines of `PUSH` and `POP` in the provided code. Also, do not change the line `BX lr`, which is used to return the PC (Program Counter) to the caller with the correct address of the code after the instructions are all finished in the function. You should be able to see the same addition results from both the C and assembly codes. If not, at least one is not right.

Note that you don't have to use `r4` and `r5` if you program carefully and wisely.

Task 3—Programming an `uint64_t` subtraction function in assembly

(20 points)

Repeat the above for the 64-bit subtraction function.

Task 4—Programming an `uint64_t` to `int64_t` two's complement conversion function in assembly

(30 points)

Here, we write a function determine the TC expression of a 64-bit number by writing an assembly code. Specifically, we determine the TC expression of `-A` where `A` is a positive number which is in the range of the positive number of `int64_t`. (You have to convince yourself that this is needed; otherwise, there can be big trouble.) We need write the assembly function to express the TC form of `-A` by using the following two standard steps:

- Take bitwise not for each bit of `A`. Note again that `A` is expressed in the `uint64_t` form.
- Add 1 to the above value. Note that we have to consider the carry from the lower 32 bits to the upper 32 bits when doing the addition. (You have to think about this carefully.)

Note that:

- We need to make sure the argument `A` is a positive number when represented using signed 64-bit integer. (Otherwise, it will violate the definition of TC expression.)
- The return value from the function should be a signed 64-bit integer. Otherwise, we cannot assign it to `-A` when it is called in C. (Of course, we can use cast to force the assignment, but we prefer not to do so.)
- When returning the 64-bit value, you need to use the number placement pattern you have observed in Task 1 to put the data results back to the appropriate registers.

Submission of your work

(10 points)

When submitting your report, please follow the assignment/submission requirements below:

- Code snippets of all your assembly instructions in the `ws6_add_sub_tc_asm_funcs` file of the project.
- The screenshot of your Task 1 where the values of r0 to r3 are shown in Keil and your conclusion as which argument go to which register(s).
- The screenshots of the `printf` results of the main function.