Computer Organization HW 2: Performance Modeling for the µRISC-V Processor

Computer Organization 2023 Programming Assignment II (toward "µRISC-V: An Enhanced RISC-V Processor Design using Spike")

Due Date: May 10, 2023 at 23:59

Overview

RISC-V processors should support a *core* ISA for integer operations, including RV32I, RV32E, RV64I, or RV128I. Additional functionality could be adopted to augment the capability of target RISC-V processors. RISC-V has a series of *standard extensions* to provide additional support beyond the core ISA, such as floating point and bit manipulation [*RISC-V ISA List*]. On the other hand, there is also a series of *non-standard extensions*, which might be specialized for certain purposes and might conflict with other extensions. If you are interested in the related contents, please refer to the document [*Extending RISC-V*].

The RISC-V Vector extension is a promising extension for the AI computing as it enables the parallel processing of mathematic operations on a RISC-V processor. The V extension involves adding a vector computation engine on the RISC-V processor, compared with the serial computing on a typical processor.

In this assignment, you will be asked to convert the given C code segments into the corresponding assembly versions, based on the skills you learned from the previous programming assignment. You will use the RISC-V V extension to implement your programs. More importantly, you will be asked to collect the performance data for your written code, and you need to use the performance data to derive the execution time of your program on the RISC-V processor based on a basic performance model, which is provided in the following section.

1. Performance Modeling

Based on your knowledge learned from **Chapter 1.6 Performance** of our course textbook (ISBN: 0128203315; ISBN-13: 9780128203316), you would derive the CPU execution time of a given program with clock cycles per instruction (CPI), instruction counts, and clock cycle time.

- The performance model used in this assignment uses cycle per instruction (CPI) to summarize the delivered performance of a RISC-V instruction executed on the target RISC-V processor, including the effect of the CPU pipeline and the memory subsystem.
- Given the above modeling concept, you will need to collect the performance data of your program to derive the CPU execution time. Specifically, you need to record the **instruction counts** of different types of RISC-V instructions.
    - The instructions are categorized into four types and their instruction counts should be recorded (accumulated) in the four counters: `arith_cnt` , `lw_cnt` , `sw_cnt` , and `others_cnt` , respectively. The table below lists the instructions and their categories.
    - The CPIs for the four types of instructions are defined in the given header files, `arith_CPI` , `lw_CPI` , `sw_CPI` , and `others_CPI` as constants. You should not alter the constant values. The related information for the CPIs is available on the table, too.
    - The derived performance data should be stored in some variables, such as `baseline_cycle_count` for the total cycle count calculated in the first execise, and `improved_cpu_time` for the CPU time computed in the second exercise.
    - NOTE: The `cycle_time` represents the clock cycle time for the target RISC-V processor. It is a constant data and its content should not be altered.
- Variables/Constants defined in the header files used in this assignment.

Var./Cons. Name	Definition
`x[ ]`	Input array 1 with size of 16 in q1_constant.h with size of 8 in q2_constant.h

Var./Cons. Name	Definition
`h[ ]`	Input array 2 with size of 16 (q1_constant.h)
`y[ ]`	Output array with size of 16 (q1_constant.h)
`tmp1[ ]`	Temp array 1 to help you finish algorithm in question 2
`tmp2[ ]`	Temp array 2 to help you finish algorithm in question 2
`arith_cnt`	used to count `add{i}, sub{i}, and{i}, or{i}, xor{i}, shift,` `vadd.vv, vadd.vx, vadd.vi vsub.vv, vsub.vx, vand.vv, vand.vx, vand.vi,` `vor.vv, vor.vx, vor.vi, vxor.vv, vxor.vx, vxor.vi, mul, vmul.vv, vmul.vx` instruction
`lw_cnt`	used to count `lw, lh, lb, li, lbu, lhu, vle8.v, vle16.v, vle32.v, vle64.v` instruction
`sw_cnt`	used to count `sw, sh, sb, vse8.v, vse16.v, vse32.v, vse64.v` instruction
`others_cnt`	used to count rest of instruction
`arith_CPI`	CPI of instructions listed in `arith_cnt`
`lw_CPI`	CPI of instructions listed in `lw_cnt`
`sw_CPI`	CPI of instructions listed in `sw_cnt`
`others_CPI`	CPI of rest of instructions
`baseline_cycle_count`	Clock cycle in `baseline()` asm volatile code you need to calculate
`improved_cycle_count`	Clock cycle in `improved()` asm volatile code you need to calculate
`cycle_time`	The given clock cycle time of the target RISC-V processor running at 2.6 GHz
`baseline_cpu_time`	The CPU time in `baseline()` you need to calculate
`improved_cpu_time`	The CPU time in `improved()` you need to calculate
`target`	Target number in question 2
`flag`	Set to 1 if the sum of two elements equals to target number in question 2

2. What Should You Do in this Assignment?

There are three exercises in this assignment. You will receive the code with the following structure.

```
CO_StudentID_HW2.zip/
└── CO_StudentID_HW2/
    |── answer.h
    ├── lab2_q1_baseline.c
    ├── lab2_q1.c
    ├── lab2_q1.h
    ├── lab2_q1_improved.c
    ├── lab2_q1_input.txt
    ├── lab2_q2_ans.c
    ├── lab2_q2.c
    ├── lab2_q2.h
    ├── lab2_q2_input.txt
    ├── makefile
    └── test
```

- The file you shold modify is below:
  - lab2_q1_baseline.c
  - lab2_q1_improved.c
  - answer.h
  - lab2_q2_ans.c

1. Baseline Code: Write Assembly Code and Report the Performance Statistics (20%)

You need to write the assembly code for the designated operations (specified in `lab2_q1.c` as shown below) with the **RV64I** ISA, and to add the four counters into the code to count the number of executed instructions. Please report the performance statistics, including the values of the four counters, total cycle count, and CPU time, of your revised program. Detailed descriptions are listed as follows.

- As shown in the `baseline()` function, you are responsible for writing the assembly for the array additions (`for (...) y[i] = h[i] + x[i];`).
  - **NOTE:** You should put your assembly code within the `lab2_q1_baseline.c` file, as indicated in `asm volatile( #include "lab2_q1_baseline.c" : [h] "+r"(p_h), ...);` within the `baseline()` function of `lab2_q1.c`.
  - There is a header file (`lab2_q1_constant.h`) specifying the constants/variables used by this assignment. It defines that the size of the arrays is set to 16. NOTE: Please do not modify the code in the header file.
- In addition, you need to insert the assembly code to **count** the number of executed instructions, according to the types of the instructions, and to store the accumulated counts in the respective counters. You will need to provide (i.e., print) the contents of the four counters (`arith_cnt`, `lw_cnt`, `sw_cnt`, and `others_cnt`), as defined in the above table.
  - You may, for example, use the following instruction to increment the content in the `lw_cnt` counter.

    > addi %[lw_cnt], %[lw_cnt], 1\n\t
- You also need to **compute** the total cycle count and the CPU execution time for the program.
  - You need to calculate `baseline_cycle_count` based on the *counter values* and the *given CPIs* (the constants defined in `lab2_q1_constant.h`). The above table defines the variables for keeping the CPI values. Please add the related code (formula) to set up the value for `baseline_cycle_count`.
  - You need to compute `baseline_cpu_time` using the *total cycle count* and the *cycle time* (the constant of the cycle time for a 2.6GHz processor defined in `lab2_q1_constant.h`). Please add the related code (formula) to compute the value for `baseline_cpu_time`.
- Your obtained scores of this exercise is determined by the correctness of your reported performance data.
  1. The four counter values, 2.5% for each counter. (10% in total)
     - `arith_cnt` (2.5%)
     - `lw_cnt` (2.5%)
     - `sw_cnt` (2.5%)
     - `others_cnt` (2.5%)
  2. The total cycle count (`baseline_cycle_count`). (5%)
  3. The CPU time (`baseline_cpu_time`). (5%)

```
"addi t2, zero, 16\n\t" // t2 = arr_size
"addi %[arith_cnt], %[arith_cnt], 1\n\t" // add arith counter
```

2. Vectorized Code: Write Assembly Code and Report the Performance Statistics (40%)

You need to re-write the assembly code (which you build in the previous exercise) with the **RISC-V V** extension or any other extension that can be supported by the installed toolchain. Similar to the previous exercise, you need to add the counters to count the executed instructions, and need to report the performance statistics, including the values of the four counters, total cycle count, and CPU time, of your vectorized program. Detailed descriptions are listed below.

- As shown in the `improved()` function, you are responsible for writing the assembly for the array additions (`for (...) y[i] = h[i] + x[i];`) using the **RISC-V V** extension [*RISC-V V Extension*]. The vectorized version would improve the execution efficiency via doing the computations in parallel.
    - **NOTE:** You should put your vectorized code within the `lab2_q1_improved.c` file, as indicated in `asm volatile( #include "lab2_q1_improved.c" :[x] "+r"(p_x), [y] "+r"(p_y), ...);` within the `improved()` function of `lab2_q1.c`.
    - The header file (`lab2_q1_constant.h`) is shared by both the `baseline()` and `improved()` functions, and should not be modified in any way.
- In addition, similar to the first exercise, you need to insert the code of the four counters (`arith_cnt` , `lw_cnt` , `sw_cnt` , and `others_cnt`) to **count** the number of executed instructions based on their instruction types. You will need to provide (i.e., print) the contents of the four counters.
- You also need to **compute** the total cycle count and the CPU execution time for the improved program.
    - You need to calculate `improved_cycle_count` based on the *counter values* and the *given CPIs* (the constants defined in `lab2_q1_constant.h`). The above table defines the variables for keeping the CPI values. Please add the related code (formula) to set up the value for `improved_cycle_count` .
    - You need to compute `improved_cpu_time` using the *total cycle count* and the *cycle time* (the constant of the cycle time for a 2.6GHz processor defined in `lab2_q1_constant.h`). Please add the related code (formula) to compute the value for `improved_cpu_time` .
- The speed up of the performance enhanced by the vectorized version against the baseline code should be presented to show the efficiency delivered by the vectorized code.
    - The speed up achieved by the vector code should be calculated and printed out by the `printf()` function, invoked at the end of the `improved()` function.
    - The speed up can be calculated with the `improved_cpu_time` and the `baseline_cpu_time` .
- Your obtained scores of this exercise is determined by the correctness of your reported performance data.
    1. The four counter values,1% for each counter. (4% in total)
        - `arith_cnt` (1%)
        - `lw_cnt` (1%)
        - `sw_cnt` (1%)
        - `others_cnt` (1%)
    2. The total cycle count (`improved_cycle_count`). (3%)
    3. The CPU time (`improved_cpu_time`). (3%)
    4. The calculated speed up (achieved by the vector version over the baseline code). (30%)
        - **Note that you can get the score only if your `baseline_cpu_time` <= 250,000 (us) and your cpu time is right.**
        - If the speedup number <= 1, you get **(0 pt)**.
        - If 1 < the speedup number <= 2, you get **(10 pt)**.
        - If 2 < the speedup number <= 4, you get **(20 pt)**.
        - If 4 < the speedup number, you get **(30 pt)**.

3. Nested For-Loop Code (60%)

You will be provided with an array of integers and a `target` number. The original C code (in `lab2_q2.c`) shows the logic of the nested loop code, where the summation of the array integers (`p_x[i]` and `p_x[j]`) is used to compare with `target`. The loop execution will break when the summation is equal to the value of the `target`. The `flag` value is used to indicate the *answer* of the given test data to see if there is a summation matching the `target`. Besides, you are asked to compute the CPU execution time of your written code. Detailed descriptions are listed below.

```
// lab2_q2.c
...
/* Original C code */
for (int i = 0; i < arr_size; i++){
    if (flag == 1)
        break;
    for (int j = 0; j < arr_size; j++){
        if ((p_x[i] + p_x[j]) == target){
            flag = 1;
            break;
        }
    }
}
...
```

- The `flag` value indicates the answer (result) of the computations.
  - If `flag` == 0, it means there is no matching found in the array elements. If `flag` == 1, it means a match is found.
- You can use the same array element more than once to obtain the summation of *array elements*.

  ```
  Example:
  arr[8] = {1, 2, 3, 4, 5, 6, 7, 8};
  target =  2;
  ```

  > `flag` should be one in this case since 1 + 1 = 2. The answer should be `Yes`.

  - The array size is fixed at eight.
- The constants and variables you need is available in the header file `lab2_q2_constant.h`.
  - You should use these constants/variables in the header file wisely (e.g., `arith_cnt`, `lw_cnt`, `sw_cnt`, `others_cnt`, and `cycle_time`) to compute the CPU time.
  - Please do not modify the code in the header file.
- You are asked to develop an *efficient* assembly code. You can provide such a code by improving the searching algorithm or by using the RISC-V V extension.
  - The *efficiency* is defined by the estimated CPU time

    > Hint: You can use `tmp1` and `tmp2` arrays wisely.

    > You can achieve O(n) computation complexity if you make good use of vector add and comparison instructions from RISC-V V extension for the implementation.
- Your obtained scores of this exercise is determined by the correctness of your reported performance data.
  1. The Yes/No answer. (20%)
  2. The CPU time for your program. (40%)
     - If 720,000 (us) < CPU time, you gets **(0%)**.
     - If 360,000 (us) < CPU time <= 720,000 (us), you gets **(10%)**.

        - If 180,000 (us) < CPU time <= 360,000 (us), you gets **(20%)**.
        - If 90000 (us) < CPU time <= 180,000 (us), you gets **(30%)**.
        - If CPU time <= 90000 (us), you gets **(40%)**.

3. Test Your assignment

We use **makefile** to judge your program. You can use the judge program to get the testing score by typing judge in your terminal.

- Get your score

  ```
  $ make score
  ```

  > If your path of proxy kernel is not `/opt/riscv/riscv64-unknown-elf/bin/pk`, change it in PK_PATH in makefile.

- Test your question 1 in lab2

  ```
  $ make lab2_q1
  $ make run_lab2_q1 TEST_DATA=1
  ```

  > TEST_DATA can change the test data which you want to test (e.g., TEST_DATA=1 ~ TEST_DATA=10).

- Test your question 2 in lab2

  ```
  $ make lab2_q2
  $ make run_lab2_q2 TEST_DATA=1
  ```

  > TEST_DATA can change the test data which you want to test (e.g., TEST_DATA=1 ~ TEST_DATA=10).

4. Submission of Your Assignment

We assume your developed code is inside the folder: `CO_StudentID_HW2` . Please follow the instructions below to submit your programming assignment.

1. Compress your source code into a `zip` file.

2. Submit your homework with NCKU Moodle.

3. The zipped file and its internal directory organization of your developed code should be similar to the example below.
    - **NOTE:** Replace all `StudentID` with your student ID number).

```
CO_StudentID_HW2.zip
└── CO_StudentID_HW2/
    ├── answer.h
    ├── lab2_q1_baseline.c
    ├── lab2_q1.c
    ├── lab2_q1.h
    ├── lab2_q1_improved.c
    ├── lab2_q1_input.txt
    ├── lab2_q2_ans.c
    ├── lab2_q2.c
    ├── lab2_q2.h
    ├── lab2_q2_input.txt
    ├── makefile
    └── test
```

!!! Incorrect format (either the file structure or file name) will lose 10 points. !!!

5. References

- RISC-V ISA List
- Extending RISC-V
- RISC-V V Extension