Computer Organization HW 1: RISC-V Assembly Programming

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

Due Date: April 12, 2023 at 23:59

Overview

This assignment is to be familiar with the hardware/software interface: instruction set architecture (ISA), as well as the environment and tools (e.g., compiler and RISC-V simulator) for RISC-V programming. In particular, this first programming assignment is to **write inline assembly code** and run the developed code on a RISC-V ISA Simulator, *Spike*, to evaluate the result.

This assignment is the fundation of building an enhanded RISC-V processor, called µRISC-V. The design of µRISC-V is evaluated with the Spike simulator.

Prerequisite

The homework assignments of this course are based on the RISC-V Spike simulator running on Ubuntu Linux (e.g., 20.04.05). As our assignments are built upon open source projects, there will be unexpected compatibility issues if you choose a different development environment.

For those students who use Mac or Windows systems, in order to save your time for establishing your development environment, our suggestion is the virtual machine based solution to work on your assignments. Please follow the procedures listed in **the HW 0 document** to establish your *virtual* environment with Oracle VirtualBox. Or, you can use some other virtual machine software (e.g., VMware's solution) to build your virtual environment. Please make sure that your tools have been installed successfully before you proceed with the follows.

Still, you might face difficulties when installing the development environment on a Ubuntu Linux system. In this case, you might like to adopt the virtual machine solution to establish the virtual environment for your programming assignments.

As will be described in the following section, there will be *three* RISC-V tools to be used in the programming assignments. After you install the required tools, the directory tree related with the tools looks like below. `$HOME` represents your home folder. `~/riscv` is the root folder of RISC-V tools and your homework folders for Computer Organization.

```
$HOME/riscv/
        ├── riscv-gnu-toolchain
        ├── riscv-pk
        ├── riscv-isa-sim
        └── CO_StudentID_HW1/
```

1. Introduction to the RISC-V Development Tools

The figure above illustrates the execution flow of a C program with Spike, a RISC-V processor simulator, on top of your host machine. There are three yellow boxes represents three different tools: *riscv-gnu-toolchain*, *riscv-isa-sim*, and *riscv-pk*.

The first software, riscv-gnu-toolchain, is the RISC-V C and C++ cross-compiler. It supports two build modes: a generic ELF/Newlib toolchain and a more sophisticated Linux-ELF/glibc toolchain. The instructions to install this software on your virtual environment are summarized in Section 1-1. You can also refer to its open source repository, RISC-V GNU Compiler Toolchain, for more information.

The second software, riscv-isa-sim, is a RISC-V simulator called *Spike* that implements a functional model of a variety of RISC-V designs. It is named after the golden spike used to celebrate the completion of the US transcontinental railway. The installation instructions are given in Section 1-2. You can find out the detailed information (e.g., the supported RISC-V extensions) in its open source repository: riscv-isa-sim.

The third software, riscv-pk, is a lightweight application execution environment that can host statically-linked RISC-V ELF binaries. It is designed to support tethered RISC-V implementations with limited I/O capability and thus handles I/O-related system calls by proxying them to a host computer. In order words, given a C program, the ordinary RISC-V instructions are emulated by Spike and the system calls (system services) are served by your virtual environment (i.e., the Ubuntu 20.02), where the system calls are forwarded by *riscv-pk* so that the underlying OS can take care of the system calls invoked by your C program. For more information, you can refer to its open source repository, riscv-pk.

2. GCC Inline Assembly

The inline assembly code provides a way to write efficient code. One of the benefits of the inline assembly is the reducing of the overheads incurred by function calls. The RISC-V C/C++ compiler is based on GCC compilers, and GCC inline assembly uses AT&T/UNIX assembly syntax. The basic format of an inline assembly code is define as below. You can see the following code example to get the feeling about the format. In order to know the detail, you would refer to the Extended Asm. from GCC HOWTO document.

```
asm volatile( AssemblerTemplate
                      : OutputOperands
                      : InputOperands
                      : Clobbers)
```

Example:

Original C program: `add.c`

```
#include <stdio.h>

int main ()
{
    int a=10, b=5;
    a = a + b;
    printf("%d\n", a);
    return 0;
}
```

Inline assembly version: `add_inline.c` (using the inline assembly code to replace the C statement: `a=a+b;`)

```
#include <stdio.h>

int main ()
{
    int a=10, b=5;
    //a = a + b;
    asm volatile(
        "add %[A], %[A], %[B]\n\t"  //AssemblerTemplate
        :[A] "+r"(a)                //OutputOperands, "=r" means write-only, "+r" 
means read/write
        :[B] "r"(b)                 //InputOperands
    );
    printf("%d\n", a);
    return 0;
}
```

- "a" is the output operand, referred to as the register %[A] and "b" is the input operand, referred to as the register %[B]. In inline assembly, you can adopt the *asmSymbolicName* syntax, which is used to rename the registers (e.g., `[A]`, `[B]`). That is, when using asmSymbolicName syntax for the output operands, you may use these names (enclosed in brackets '[]', %[A]), instead of digits (e.g., %1). The benefit for the asmSymbolicName syntax is more readable and more maintainable since reordering index numbers is not necessary when adding or removing operands.

- "r" is a constraint on the operands. "r" says to GCC to use any register for storing the operands. The constraint modifier "=" says the output operand is write-only. The constraint modifier "+" says the output operand can both read and write.

- IMPORTANT! At the end of each instruction, you should add the string `\n\t`.

Please refer to "**Test the RISC-V tools**" in the **HW 0 document** for the procedures to compile the C program with the RISC-V compiler and to run the executable with the RISC-V simulator (Spike).

3. What Should You Do in This Assignment?

You are asked to convert the given C code segments into the corresponding assembly versions. In particular, there are four C programs from **3-1** to **3-4** in this assignment.

As for **3-1** and **3-2**, you should use the C programs *belonging to your graoup* as the input C programs.

As for **3-3** and **3-4**, all of you need to convert the given matrix multiplication code `matrix1x3.c` into the corresponding assembly versions, respectively.

To find your *group ID* for **3-1** and **3-2**, you shoud do the modulo operation. The remainder of YOUR_STUDENT_ID divided by 3 is your group ID. The calculations to determine your group ID are listed below.

- You belong to Group 1 if (YOUR_STUDENT_ID mod 3) == 1.
- You belong to Group 2 if (YOUR_STUDENT_ID mod 3) == 2.
- You belong to Group 3 if (YOUR_STUDENT_ID mod 3) == 0.

3-1 A basic program (30%)

In 3-1, you will be given a simple C program (`sub_inline.c`, `mul_inline.c`, or `div_inline.c`, according to your *group ID*), and you need to write the inline assembly code to perform the specific operation (subtraction, multiplication, or division according to your *group ID*) on the two input integer numbers.

That is, given the two input interger numbers stored in *a* and *b*, you should write the inline assembly to obtain the result of the computation `a = a op b;`, where `op` is determined by your *group ID*.

Your revised C program (with the inline assembly) will be tested by evaluating the printed results against the randomly generated input numbers.

The C programs of the respective group are listed as below.

- Group 1: sub_inline.c

```
// description: a = a - b

#include <stdio.h>

int main ()
{
    int a, b;
    scanf("%d %d", &a, &b);
    //a = a - b;
    asm volatile(/*Your Code*/);
    printf("%d\n", a);
    return 0;
}
```

- Group 2: mul_inline.c

```
// description: a = a * b

#include <stdio.h>

int main ()
{
    int a, b;
    scanf("%d %d", &a, &b);
    //a = a * b;
    asm volatile(/*Your Code*/);
    printf("%d\n", a);
    return 0;
}
```

- Group 3: div_inline.c

```
// description: a = a / b

#include <stdio.h>

int main ()
{
    int a, b;
    scanf("%d %d", &a, &b);
    //a = a / b;
    asm volatile(/*Your Code*/);
    printf("%d\n", a);
    return 0;
}
```

3-2 An array program (30%)

3-2 is similar to 3-1, except that there are multiple input integers kept in integer arrays *a* and *b*, and the calculated outputs are stored in the array *c*.

In 3-2, you should write the inline assembly to accomplish the operations specified in the original C code segment of your C program (determined by your *group ID*; `subvector_inline.c` for Group 1, `mulvector_inline.c` for Group 2, and `divvector_inline.c` for Group 3).

Your written code will be evaluated by the printed outputs against the given input data.

- Group 1: subvector_inline.c

```
// description: cn = an - bn

#include <stdio.h>

int main ()
{
    int a[10] = {0}, b[10]= {0}, c[10] = {0};
    int i, arr_size = 10;
    for(i = 0; i<arr_size; i++) scanf("%d", &a[i]);
    for(i = 0; i<arr_size; i++) scanf("%d", &b[i]);
    for(i = 0; i<arr_size; i++) scanf("%d", &c[i]);

    int *p_a = &a[0];
    int *p_b = &b[0];
    int *p_c = &c[0];
    /* Original C code segment
    for (int i = 0; i < arr_size; i++){
        *p_c++ = *p_a++ - *p_b++;
    }
    */
    for (int i = 0; i < arr_size; i++)
        asm volatile(/*Your Code*/);


    p_c = &c[0];
    for (int i = 0; i < arr_size; i++)
        printf("%d ", *p_c++);


    return 0;
}
```

- Group 2: mulvector_inline.c

```
// description: cn = an * bn

#include <stdio.h>

int main ()
{
    int a[10] = {0}, b[10]= {0}, c[10] = {0};
    int i, arr_size = 10;
    for(i = 0; i<arr_size; i++) scanf("%d", &a[i]);
    for(i = 0; i<arr_size; i++) scanf("%d", &b[i]);
    for(i = 0; i<arr_size; i++) scanf("%d", &c[i]);

    int *p_a = &a[0];
    int *p_b = &b[0];
    int *p_c = &c[0];

    /* Original C code segment
    for (int i = 0; i < arr_size; i++){
        *p_c++ = *p_a++ * *p_b++;
    }
    */
    for (int i = 0; i < arr_size; i++)
        asm volatile(/*Your Code*/);


    p_c = &c[0];
    for (int i = 0; i < arr_size; i++)
        printf("%d ", *p_c++);


    return 0;
}
```

- Group 3: divvector_inline.c

```
// description: cn = an / bn

#include <stdio.h>

int main ()
{
    int a[10] = {0}, b[10]= {0}, c[10] = {0};
    int i, arr_size = 10;
    for(i = 0; i<arr_size; i++) scanf("%d", &a[i]);
    for(i = 0; i<arr_size; i++) scanf("%d", &b[i]);
    for(i = 0; i<arr_size; i++) scanf("%d", &c[i]);

    int *p_a = &a[0];
    int *p_b = &b[0];
    int *p_c = &c[0];

    /* Original C code segment
    for (int i = 0; i < arr_size; i++){
        *p_c++ = *p_a++ / *p_b++;
    }
    */
    for (int i = 0; i < arr_size; i++)
        asm volatile(/*Your Code*/);


    p_c = &c[0];
    for (int i = 0; i < arr_size; i++)
        printf("%d ", *p_c++);


    return 0;
}
```

3-3 A matrix multiplication program (30%)

In 3-3, all of the students should work on the same C program `matrix1x3.c` to re-write the assembly code.

matrix1x3.c

```
/*
 * description:         1x3 matrix - multiply benchmarking
 *
 *               |h11 h12 h13|   |x1|   |y1|   | h11*x1+h12*x2+h13*x3 |
 *               |h21 h22 h23| * |x2| = |y2| = | h21*x1+h22*x2+h23*x3 |
 *               |h31 h32 h33|   |x3|   |y3|   | h31*x1+h32*x2+h33*x3 |
 *
 * Element are to store in following order:
```

```
 *
 * matrix h[9]={h11,h12,h13, h21,h22,h23, h31,h32,h33}
 * vector x[3]={x1,x2,x3}
 * vector y[3]={y1,y1,y3}
 */
#include<stdio.h>

int main()
{
  int f,i=0;
  int h[9]={0}, x[3]={0}, y[3]={0};
  for(i = 0; i<9; i++) scanf("%d", &h[i]);
  for(i = 0; i<3; i++) scanf("%d", &x[i]);
  for(i = 0; i<3; i++) scanf("%d", &y[i]);

  int *p_x = &x[0] ;
  int *p_h = &h[0] ;
  int *p_y = &y[0] ;

  /* Level 1 for loop */
  for (i = 0 ; i < 3; i++)
    {
      /* p_x points to the beginning of the input vector */
      p_x = &x[0]  ;

      /* Level 2 for loop */
      /* do matrix multiply */
      for (f = 0 ; f < 3; f++)
        *p_y += *p_h++ * *p_x++ ;

      /* next element */
      p_y++ ;
    }

  p_y = &y[0];
  for(i = 0; i<3; i++)
    printf("%d \n", *p_y++);

  return(0)  ;

}
```

Given the above matrix multiplication code, which consists of two-level for loop, please convert the Level-2 loop body (`*p_y += *p_h++ * *p_x++;` at line #37) into the inline assembly version. The example of your inline assembly version will look like the below code. Hint: Please provide your assembly code in `asm volatile()`, as what you did in the 3-1 and 3-2.

```
// description: matrix muliply with two-level for loop

#include<stdio.h>

```

```
int main()
{
  int f,i=0;
  int h[9]={0}, x[3]={0}, y[3]={0};
  for(i = 0; i<9; i++) scanf("%d", &h[i]);
  for(i = 0; i<3; i++) scanf("%d", &x[i]);
  for(i = 0; i<3; i++) scanf("%d", &y[i]);

  int *p_x = &x[0] ;
  int *p_h = &h[0] ;
  int *p_y = &y[0] ;

  for (i = 0 ; i < 3; i++)
    {

      p_x = &x[0]  ;

      for (f = 0 ; f < 3; f++)
        asm volatile(/*Your Code*/);

    }

  p_y = &y[0];
  for(i = 0; i<3; i++)
    printf("%d \n", *p_y++);

  return(0)  ;

}
```

3-4 Matrix Multiply (30%)

In 3-4, all of the students should refer to the C program`matrix1x3.c` in 3-3, and convert the *two-level loop nest* into the assembly code. Note that you should use RISC-V assembly to implement the two-level loops within `asm volatile()`; that is, your implemented code segment CANNOT include the C based keywords (e.g., *for* or *while*).

```
// description: matrix muliply without for loop

#include<stdio.h>

int main()
{
  int i=0;
  int h[9]={0}, x[3]={0}, y[3]={0};
  for(i = 0; i<9; i++) scanf("%d", &h[i]);
  for(i = 0; i<3; i++) scanf("%d", &x[i]);
  for(i = 0; i<3; i++) scanf("%d", &y[i]);

  int *p_x = &x[0] ;
  int *p_h = &h[0] ;
  int *p_y = &y[0] ;

  asm volatile(/*Your Code*/);

  p_y = &y[0];
  for(i = 0; i<3; i++)
    printf("%d \n", *p_y++);

  return(0)  ;

}
```

4. Test Your Assignment

We use local-judge to judge your program, where you can install the local-judge via the command: `pip3 install local-judge` on Ubuntu systems using the Python3 system. You need to install Python3 first, and you can use the following commands to install and run the `judge` program against your assignment.

```
$ sudo apt install python3-pip
$ pip3 install local-judge
$ cd ~/riscv/CO_StudentID_HW1
$ judge
```

The example output of your program judged by `judge` is shown as below.

```
$ cd ~/riscv/CO_StudentID_HW1
$ judge
=======+=============================================================
Sample | Accept
=======+=============================================================
     1 | ✔
=======+=============================================================
     2 | ✔
=======+=============================================================
     3 | ✔
=======+=============================================================
     4 | ✗
=======+=============================================================
Total score: 90
```

5. Submission of Your Assignment

We assume your developed code is inside the folder: `CO_StudentID_HW1`, with the directory structure given in the above "**Prerequisite**" section. 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_HW1.zip
└── CO_StudentID_HW1/
    ├── 1.c
    ├── 2.c
    ├── 3.c
    └── 4.c
```

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

5. Reference

- GCC-Inline-Assembly-HOWTO
- Extended Asm - Assembler Instructions with C Expression Operands
- RISC-V Assembly Programmer's Manual
- The RISC-V Instruction Set Manual Version 2.2