tags `Computer Organization`

Computer Organization HW 0: Environmental Setup

Overview

This comprehensive guide serves as a thorough resource for preparing the experimental environment and GNU GCC Extended ASM (inline assembly) required to complete the programming assignments in the Computer Organization course.

In particular, the procedures for installing the required tools listed below will be provided.

1. GNU toolchain for RISC-V (commit hash a33dac0),
2. RISC-V ISA simulator (Spike; commit hash bbaec51), and
3. Proxy kernel (commit hash e5563d1).

The GNU toolchain, including the GNU Compiler Collection (GCC), is used to compile C source code into executable files for RISC-V machines.
The RISC-V ISA simulator (Spike) emulates a RISC-V environment, while the Proxy kernel facilitates system call redirection.

You can establish the development environment by one of the two methods below.

1. Install the prebuilt virtual machine image, which contain the virtual environment for developing the programming assignments in this course.
2. Build the above three tools from their source codes.

If you decide to work within the virtualized environment without building the software from scratch, please refer to "**A. Install the pre-built environment**" for further instructions. The prebuilt environment runs on Ubuntu Linux 24.04.1 with the tools installed.

Alternatively, if you choose to build from the source codes, you will need to clone the source projects directly from their respective websites and proceed with the build process from scratch. Please refer to "**B. Build from scratch**" for detailed instructions.

Upon installing the tools, please refer to the guide titled "**C. Test the RISC-V tools**" to verify their successful installation.

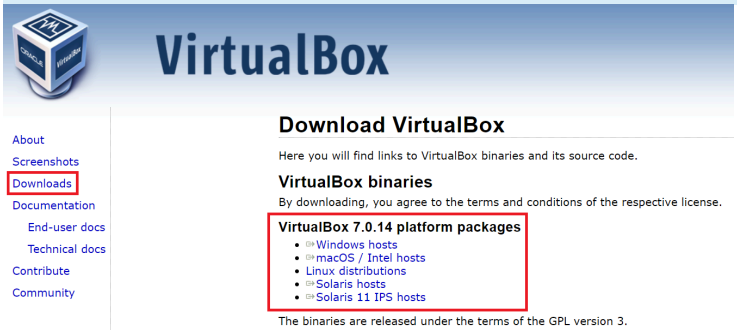
Finally, "**D. Practice for Programming Assignments**" provides a tutorial on GNU GCC inline assembly along with corresponding exercises. Students are encouraged to practice these exercises as they progress through the course. These exercises are **NOT** graded.

A. Install the pre-builded environment

I. Install VirtualBox (the virtual machine software) Download Link

Select, download, and install the appropriate version (Windows, Linux, or macOS/Intel hosts) of the software on your host machine from the **Download Link**, as indicated in the image below.

We do not offer an arm64/aarch64 version of the prebuilt virtual machine image.
Students with Apple Silicon devices are suggested to utilize x86 computers in the Computer Classrooms of the Computer and Network Center.



II. Import the prebuilt virtual machine image

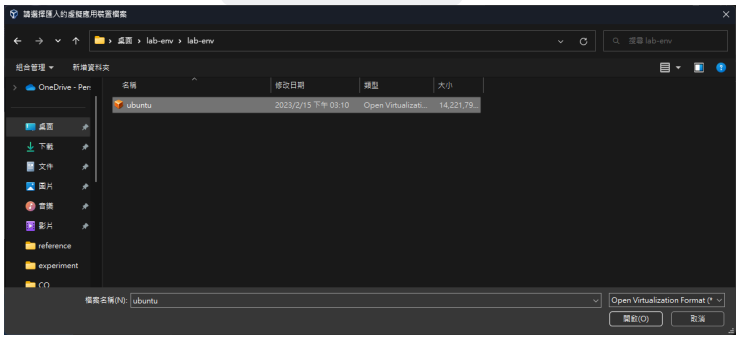
Download and decompress the prebuilt image. Afterwards, import the decompressed image into VirtualBox.
The following steps outline the process of initiating the virtualized Ubuntu environment:

username: CompOrg
password: nckucsie

1. Download the prebuilt image `CompOrg2025.ova` from any available mirror on Moodle.
2. Launch VirtualBox
3. Click `Import` button

4. Click the `Browse file` button

5. Choose the file: `CompOrg2025.ova` from the selected folder

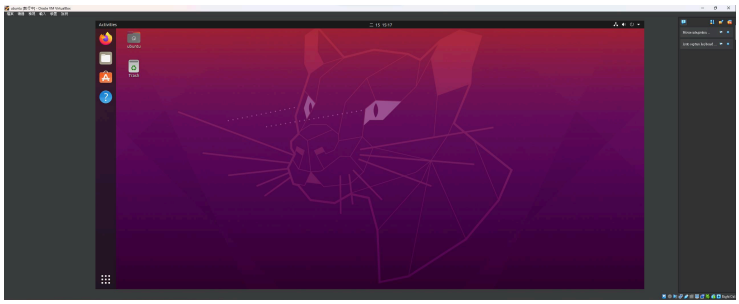


6. Click `Finish` button

7. Wait for VirtualBox to load the virtual environment

8. On the sidebar, you will find the **CompOrg2025** virtual machine of the prebuilt environment.
   This indicates the prebuilt environment has been successfully imported.
   Click `Start` to power on the virtual machine

9. A new window displays the desktop environment of the Ubuntu system.
   This indicates that you have successfully initiated the Ubuntu Linux system.



B. Build from scratch

I. Ubuntu installation

The following instructions are based on the Ubuntu Linux (24.04.1) environment, other system may have different commands or procedures, depends on the package manager of the distribution.

On MacOS and Windows, the default filesystem is case-insensitive. Building glibc on such filesystem is not supported. **YOU ARE HIGHLY RECOMMENDED** to work with a Linux environment.
YOU ARE ALSO DISCOURAGED TO CONVERT THE FILE SYSTEM TO CASE-SENSITIVE, as it may corrupt the system and cause the system to be unbootable.
ALL RISK IS ON YOUR OWN.
You may refer to the Virtual Machine creation guide, and the Ubuntu installation guide for installing Ubuntu on a physical machine or virtual machine.

At this point, we assume that you have a working Ubuntu Linux environment. Please proceed with the following instructions to install the three RISC-V tools

II. Install GNU Toolchain for RISC-V (GCC tools)

1. Create a package folder under the `/opt` folder (i.e., `/opt/riscv`). Subsequently, add the `/opt/riscv` to the environment variable `$RISCV`. Afterwards, create a sub-directory named `bin` within your package directory (i.e., `/opt/riscv/bin`). Finally, add the `/opt/riscv/bin` to the environment variable `$PATH`. The relevant commands are provided below.

   ```
   $ sudo mkdir /opt/riscv && sudo mkdir /opt/riscv/bin
   $ echo 'export RISCV=/opt/riscv' >> ~/.bashrc
   $ echo 'export PATH=$PATH:$RISCV/bin' >> ~/.bashrc
   $ source ~/.bashrc
   ```

2. After that, you should create a project directory (e.g., `$HOME/riscv`).

   ```
   $ mkdir ~/riscv
   ```

3. Run the following commands to Install the tool:

   - Retrieve and install the necessary packages using the `sudo apt install` command.
   - Clone the latest version of the `riscv-gnu-toolchain` project repository from GitHub using the `git clone` command.
   - Execute the `./configure` command to set up the environment variables before building the project.

- To construct and install the project in parallel, utilize the `sudo make linux -j$(nproc)` command. This command will utilize the number of logical threads available on your machine to execute the build process concurrently.

```
    $ cd ~/riscv
    $ sudo apt update
    $ sudo apt install autoconf automake autotools-dev curl python3 python3-pip
libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf
libtool patchutils bc zlib1g-dev libexpat-dev ninja-build git cmake libglib2.0-dev
    $ git clone https://github.com/riscv/riscv-gnu-toolchain
    $ cd riscv-gnu-toolchain
    $ ./configure --prefix=$RISCV --enable-multilib
    $ sudo make linux -j$(nproc)
```

4. After the above operations, the **riscv-gnu-toolchain** will be installed under the path: `$RISCV/riscv-gnu-toolchain` and the **riscv64-unknown-linux-gnu** packages will be installed under the path: `$RISCV`.
   The following command is used to show the version information of the installed GCC compiler (the expected output message is shown below.)

   ```
   $ riscv64-unknown-linux-gnu-gcc -v
   riscv64-unknown-linux-gnu-gcc (g04696df09) 14.2.0
   ```

III. Install the RISC-V ISA simulator (Spike)

The following commands are used to install the simulator and are similar to those shown above.

Please ensure that you have changed working directory to `$HOME/riscv` (e.g., `cd ~/riscv/`) before running the following commands.

The commands below build the Spike simulator in the folder `~/riscv/riscv-isa-sim/build`, and Spike will be installed under the path: `$RISCV/bin`.

```
$ cd ~/riscv
$ sudo apt install device-tree-compiler libboost-regex-dev libboost-all-dev
$ git clone https://github.com/riscv/riscv-isa-sim.git
$ cd riscv-isa-sim
$ mkdir build
$ cd build
$ ../configure --prefix=$RISCV
$ make
$ sudo make install
```

The veracity of the Spike installation can be validated through the instructions outlined in "**Test the RISC-V tools**".

IV. Install Proxy Kernel

The proxy kernel serves as a mediator between the I/O system of Spike and the host machine. Its primary function is to redirect I/O operations, enabling the emulation and execution of system calls within the virtualized RISC-V

environment. This is crucial because Spike lacks an external I/O device simulation, necessitating the presence of the proxy kernel to facilitate proper I/O operations.

The commands below build the PK in the folder `~/riscv/riscv-pk`, and Proxy Kernel will be installed under `$RISCV/riscv64-unknown-linux-gnu/bin`.

```
$ cd ~/riscv
$ git clone https://github.com/riscv/riscv-pk.git
$ cd riscv-pk
$ mkdir build
$ cd build
$ ../configure --prefix=$RISCV --host=riscv64-unknown-linux-gnu --with-
arch=rv64gc_zifencei
$ make
$ sudo make install
```

Please verify that the RISC-V toolchain architecture in $RISCV/bin is consistent with the value specified in the `--host` flag.

The veracity of the PK installation can be ascertained through the instructions provided in the section entitled "**C. Validate the RISC-V Toolchain Installation**".

C. Validate the RISC-V Toolchain Installation

To confirm that the required tools are correctly installed and meet the specifications for future assignments, you should perform the following test.

1. Clone the Homework 0 repository

```
$ git clone https://github.com/ASRLabCourses/CompOrg2025_HW0
```

2. Run the toolchain test suite:

```
$ cd CompOrg2025_HW0/rv_toolchain_test
$ make all
```

If the setup is correct, the output should resemble the following:

```
make: [Makefile:24: clean] Error 1 (ignored)
riscv64-unknown-linux-gnu-gcc -static -march=rv64gcv -o hello hello.c
spike --isa=RV64GCV /opt/riscv/riscv64-unknown-linux-gnu/bin/pk hello
Hello World from RISC-V!
riscv64-unknown-linux-gnu-gcc -static -march=rv64gcv -o vadd vadd.c
spike --isa=RV64GCV /opt/riscv/riscv64-unknown-linux-gnu/bin/pk vadd
p_a: 1, 1, 1, 1, 1, 1, 1, 1,
p_b: 0, 1, 2, 3, 4, 5, 6, 7,
p_c: 1, 2, 3, 4, 5, 6, 7, 8,
```

D. Practice for Programming Assignments

The programming assignments in this course use GCC Inline Assembly to develop RISC-V assembly codes within a C program. The following paragraphs introduce the concept of GCC Inline Assembly and then give programming exercises for writing simple programs with RISC-V assembly. Finally, you can be familiar with the local-judge tool to check the correctness of your written code.

I. GCC Inline Assembly

The inline assembly code provide 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 a high-level concept of `the format of inline assembly`. For detailed information, you may refer to Extended asm. from GCC Online Documents.

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

Example:

Original C program: `add.c`

```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`

```c
#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" indicates read/write
        :[b] "r"(b)  // InputOperands
    );
    printf("%d\n", a);
    return 0;
}
```

- "a" is the output operand, referred to by the register %[a] and "b" is the input operand, referred to by the register %[b]. In inline assembly, you can adopt the `asmSymbolicName syntax` method which is to renaming the registers (e.g., `[a], [b]`). The benefit for the asmSymbolicName syntax method 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.
- Beware! Every line of instruction should be ended with `\n\t`.

II. Programming Exercise

The source files for these exercises are included in the Homework 0 GitHub repository. You may skip the following command if you have already cloned the repository in "**C. Validate the RISC-V Toolchain Installation.**"

```
$ cd ~
$ git clone https://github.com/ASRLabCourses/CompOrg2025_HW0
```

Ensure that the `judge*.conf` files are placed in the same directory as your code (e.g., `$HOME/CompOrg2025_HW0/CO_StudentID_HW0`). The directory structure for this homework is shown below:

```
CompOrg2025_HW0/
├── CO_StudentID_HW0
|   ├── arith.c
|   ├── array_arith.c
|   ├── Makefile
|   ├── judge1.conf
|   └── judge2.conf
├── README.md
├── answer
|   ├── arith.c
|   └── array_arith.c
├── rv_toolchain_test
|   ├── Makefile
|   ├── hello.c
|   └── vadd.c
└── testcases
    ├── expected
    |   ├── 1.out
    |   └── 2.out
    └── input
        ├── 1.txt
        └── 2.txt
```

Make sure that the directory path **CO_StudentID_HW0** does not contain any spaces.

Exercise 1: Arithmetic Operation

Complete the following code by writing your inline assembly implementation in the C file:

- CompOrg2025/CO_StudentID_HW0/arith.c

```c
#include <stdint.h>
#include <stdio.h>

int main()
{
    int32_t a, b;
    FILE *input = fopen("../testcases/input/1.txt", "r");
    fscanf(input, "%d %d", &a, &b);
    fclose(input);
    /* a = a - b */
    asm volatile(/* Your Code */);
    printf("%d\n", a);
    return 0;
}
```

Exercise 2: Array

Complete the following code by writing your inline assembly implementation in the C file:

- CompOrg2025/CO_StudentID_HW0/array_arith.c

```c
#include <stdint.h>
#include <stdio.h>

int main()
{
    int32_t a[10] = {0}, b[10] = {0}, c[10] = {0};
    int32_t i, arr_size = 10;
    FILE *input = fopen("../testcases/input/2.txt", "r");
    for (i = 0; i < arr_size; i++)
        fscanf(input, "%d", &a[i]);
    for (i = 0; i < arr_size; i++)
        fscanf(input, "%d", &b[i]);
    fclose(input);
    int32_t *p_a = a;
    int32_t *p_b = b;
    int32_t *p_c = c;
    /* Original C code segment
    for (int32_t i = 0; i < arr_size; i++)
        *p_c++ = *p_a++ - *p_b++;
    */
    asm volatile(/* Your Code */);
    p_c = c;
    for (int32_t i = 0; i < arr_size; i++)
        printf("%d ", *p_c++);
    printf("\n");
    return 0;
}
```

III. Test Your Program

We use local-judge to judge your program. You can use the command `pip3 install local-judge` to download and install the local-judge, as shown below.

```
$ sudo apt install python3-pip    # Install pip tool
$ pip3 install --break-system-packages local-judge    # Install local-judge via pip3
$ echo "export PATH="$PATH":/home/CompOrg/.local/bin" >> ~/.bashrc    # Add path for local-judge
$ source ~/.bashrc
```

Now, you can use the judge program to get the score of your developed code with the following commands.

```
$ cd ~/riscv/CO_StudentID_HW0
$ make judge
=======+============================================================
Sample | Accept
=======+============================================================
     1 | ✔
=======+============================================================
     2 | ✘
=======+============================================================
Obtained/Total scores: 50/100
```

If you want to judge a specific exercise, you can use the command `judge -c [CONFIG]`. For example, you can check the result of the first exercise via the command `judge -c judge1.conf`.

```
$ cd ~/riscv/CO_StudentID_HW0
$ judge -c judge1.conf
=======+============================================================
Sample | Accept
=======+============================================================
     1 | ✔
=======+============================================================
Correct/Total problems: 1/1
Obtained/Total scores: 50/50
```

If your code generates wrong output, you can use the parameter `-v 1` while running the judge command to check the differences between your output and the correct answer. The example message is shown as below.

```
Example:
standard answer is 5.
your program output is 15.
```

```
$ cd ~/riscv/CO_StudentID_HW0
$ judge -c judge1.conf -v 1
=======+============================================================
Sample | Accept
=======+============================================================
     1 | ✘
=======+============================================================
diff --git a/home/ubuntu/riscv/answer/1.out b/../output/1local_1709739995.out
index 7ed6ff8..60d3b2f 100755
--- a/home/ubuntu/riscv/answer/1.out
+++ b/../output/1local_1709739995.out
@@ -1 +1 @@
515

=======+============================================================
Correct/Total problems: 0/1
Obtained/Total scores:  0/50
```