Lab Experiment- Introduction to RISCV Assembly Programming

Objective:

The objective of this lab is to gain practical experience in writing RISC-V assembly code, running it on Spike, and using Spike's debugging features.

Prerequisites:

- RISC-V GNU Toolchain (riscv64-unknown-elf-gcc, riscv64-unknown-elf-as)
- Spike RISC-V ISA Simulator
- Text editor
- Sample codes: sample_src.S, linker.ld

Task 1: Installing spike and riscv toolchain

Steps:

1. Configure the riscv toolchain

```
sudo apt-get install gcc-riscv64-unknown-elf
```

2. Run the following bash script to download and install spike

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

3. Add spike to the environment path by writing the following script (update path according to your machine)

```
export PATH=$PATH:/opt/riscv/bin
```

Task 2: Running a basic example on Spike

Steps

1. Create a linker script: Create a file named "**link.ld**" with the following content:

```
OUTPUT_ARCH( "riscv" )
ENTRY( _start )

SECTIONS
{
  . = 0x80000000;
  .text : { *(.text) }
  .data : { *(.data) }
  .bss : { *(.bss) }
  .tohost : { *(.tohost) }
}
```

This script tells the linker to place your code starting at address 0x80000000, which is a valid starting address for Spike.

2. Create an assembly file with name "**example.S**" and write the following content

```
.global _start

.section .text
_start:
    # Any code here
    li a0, 0 # Initialize counter
    li a1, 10 # Set maximum count
loop:
     addi a0, a0, 1 # Increment counter
     blt a0, a1, loop  # Loop if counter < max

    # Code to exit for Spike (DONT REMOVE IT)
    li t0, 1
    la t1, tohost
    sd t0, (t1)
```

```
    # Loop forever if spike does not exit
1:  j 1b

.section .tohost
.align 3
tohost: .dword 0
fromhost: .dword 0
```

3. Assemble and link your code: Use these commands to assemble and link your code with the new linker script:

```
riscv64-unknown-elf-as -o example.o example.s
riscv64-unknown-elf-ld -T link.ld -o example example.o
```

4. Run your code with Spike:

```
spike example
```

5. For debugging

```
spike -d example
```

6. Or you may use the following command to see the result too:

```
spike -d –log-commits example
```

7. You can then use debugging commands like:

```
(spike) until pc 0x80000000
(spike) r
(spike) s
(spike) mem 0x80000000 +32
```

8. If you want to use HTIF for output, modify your code like this:

```
.global _start

.section .text
_start:
    li t0, 0x10000000  # HTIF base address
    la t1, message     # Load address of message

print_loop:
    lb t2, (t1)        # Load byte from message
    beqz t2, done      # If byte is zero, exit loop
    sw t2, 0(t0)       # Write byte to HTIF
```

```
    addi t1, t1, 1      # Move to next byte
    j print_loop

done:
    # Signal test pass to Spike
    li t0, 1
    la t1, tohost
    sd t0, (t1)

    # Loop forever
1:  j 1b

.section .data
message:
    .string "Hello, World!\n"

.section .tohost
.align 3
tohost: .dword 0
fromhost: .dword 0
```

Exercise:

For each exercise in this lab manual, follow these steps:

- Write your RISC-V assembly code using the provided template.
- Use the Makefile to assemble, link, and run your code.
- Debug your code using Spike when necessary.
- Submit your work using the Makefile.

MakeFile for this exercise can be defined as follows:

```
# Makefile for RISC-V Assembly Exercises

# Compiler and emulator
AS = riscv64-unknown-elf-as
LD = riscv64-unknown-elf-ld
SPIKE = spike
```

```
# Default target
all: $(PROG)

# Rule to assemble and link
$(PROG): $(PROG).s
$(AS) -o $(PROG).o $
$(LD) -T linker.ld -o $@ $(PROG).o

# Rule to run with Spike
run: $(PROG)
$(SPIKE) $(PROG)

# Rule to debug with Spike
debug: $(PROG)
$(SPIKE) -d –log-commits $(PROG)

# Clean up
clean:
rm -f *.o $(PROG)

.PHONY: all run debug clean
```

Problems:

1. Implement a program to calculate the absolute difference between two numbers.
2. Implement a function to count the number of set bits in a 32-bit word.
3. Implement a program to calculate the factorial of a number.
4. Implement a program to reverse an array in-place.
5. Implement an insertion sort algorithm for sorting an array.

Tasks:

- Write an assembly program for restoring division algorithm in RISC-V assembly language.
    - Use the toolchain to build the assembly file from your C file.
    - Compare the two assembly files. Which is more optimized?

  - Run both on spike and see their working.
- Write an assembly program for setting or clearing any bit in a 32-bit number in RISC-V assembly language.
  - Write a C code for the same purpose.
  - Use the toolchain to build the assembly file from your C file.
  - Compare the two assembly files. Which is more optimized?
  - Run both on spike and see their working.
- Write an assembly program for non-restoring 32-bit unsigned division in RISC-V assembly language.
  - Write a C code for the same purpose.
  - Use the toolchain to build the assembly file from your C file.
  - Compare the two assembly files. Which is more optimized?

  Run both on spike and see their working