A10 RISC-V系統模擬器驗證分析

C1: RISC-V Tool Chain

C2: RSIC-V Add Custom Instruction

C3: RISC-V Profiling

C1: RISC-V Tools And System Simulator

Outline

- Introduction to Toolchain
  - Compiler
  - Linker
  - Library & Debugger
- RISCV-Simulator
  - Spike
  - QEMU
  - Gem5
- Lab

What is toolchain?

Toolchain is a set of programming tools. A basic toolchain include:

- Compiler
  - Compiler is a tool that translate "source code"(written by programming language) into "target language".
- Linker
  - Linker can link "target file"(from compiler) and "libraries" together and generate an executable file.
- Library
  - Libraries is a collection of sub-functions that already compiled. Provide service to other program.
- Debugger
  - To test and debug the target programs.

Note: For different machine(CPU) need different toolchain. Because CPU has many different type and commands.

Introduction

- The advantage to use the simulator:
  - Complete computer architecture without having a hardware
  - Reduce time taken by development
  - Get more data between the variables to improve the hardware design
  - Compare with the different architectures and find the best one to keep the cost down
  - Enable to simulate complicated system whether it is exist or not
- Simulator
  - Spike - golden reference simulator
  - QEMU - open source full-system simulator
  - Gem5 - modular platform simulator

Step1: Set environment variable & PATH

- Setting variables "RISCV" and "PATH".
- "RISCV" is a path that you want to create toolchain.

```
$ echo "export RISCV=/path/to/install/riscv/toolchain">> ~/.bashrc
$ echo "export PATH=$RISCV/bin:$PATH" >> ~/.bashrc
$ source ~/.bashrc
```

Note: You can reopen terminal instead enter the command "source ~/.bashrc".

- You will see the text added in the end of ".bashrc"

```
if ! shopt -oq posix; then
  if [ -f /usr/share/bash-completion/bash_completion ]; then
    . /usr/share/bash-completion/bash_completion
  elif [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
  fi
fi
export RISCV=/home/riscv/RISCV
export PATH=/home/riscv/RISCV/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin……
```

Step2: Build RISC-V Toolchain

- Start building toolchain

```
$ cd riscv-gnu-toolchain
$ ./configure --prefix=$RISCV
$ make -j3
```

Note : the number of make –j(N+1) is base on your CPU cores N

- Using virtual machine or less cores will spend more time on this step.
- After the process you will see the result:

```
make[3] Leaving directory '/home/tf/riscv-tools/riscv-gnu-toolchain/build-gcc newlib-stage2/gcc '
make[2]: Leaving directory '/home/tf/riscv-tools/riscv-gnu-toolchain/build-gcc- newlib-stage2 '
make[1]: Leaving directory '/home/tf/riscv-tools/riscv-gnu-toolchain/build-gcc- newlib-stage2 '
mkdir -p stamps/ && touch stamps/build-gcc-newlib-stage2
```

Step3: Build Simulate Environment

- Build a simulate environment base on this RISC-V CPU by Spike.

```
$ cd .. (back to riscv-tools)
$ ./build-spike-pk.sh
```

- After waiting you may see this result:

```
Installing project riscv-isa-sim
mkdir /home/tf/riscv/include/fesvr
mkdir /home/tf/riscv/lib/pkgconfig
```

```
Installing project riscv-pk
mkdir /home/tf/riscv/riscv64-unknown-elf/include/riscv-pk
mkdir /home/tf/riscv/riscv64-unknown-elf/lib/riscv-pk

RISC-V Toolchain installation completed!
```

Common Workflow (1/2)

- Add .c file at any location, then compile it with riscv64-unknown-elf-gcc or riscv64-unknown-elf-g++

```
//hello.c
#include <stdio.h>
int main(){
        printf("Hello World!!\n");
        return 0;
}
```

```
$ riscv64-unknown-elf-gcc hello.c -o hello
```

- Use -o to specify the name of the output binary file

```
riscv@riscv-VirtualBox:~$ riscv64-unknown-elf-gcc hello.c -o hello
riscv@riscv-VirtualBox:~$ dir
hello.c   hello
```

Common Workflow (2/2)

- After compilation, we want to know verify the correctness of our program
- Run the compiled program with Spike and you can see the result

```
$ spike pk hello
bbl loader
Hello World!!  ← Result
```

- Spike also support debug mode like gdb

C2: Add Instructions

Outline

- Benefits of adding custom instructions
- Adding Custom Instruction to RISC-V
  - Introduce of workflow
  - Verify the result
- Adding Custom Instructions on software
  - Basic Workflow
  - Verify the result

Benefits of adding custom instructions

- Custom instructions are a key value proposition of RISC-V.
- The key challenge in here is to optimize instructions.
- In real design, flow for optimizing custom instructions in RISC-V processors is being used.
- In some specific case, using custom instructions can boost the performance of RISC-V.

Common Workflow

- First, we need to recognize the steps of adding instruction. Basically we are target our specific program. And add an unique instruction for it.
  - **1.Decide instruction**
    - Define the type and opcode
  - **2.modify .v files**
    - This step is about to introduce how to modify and what .v files we are going to modify.
  - **3. Check control signal**
    - There might be some changes in control signals, we need to check it to see if it's right.
  - **4. Test new instruction**
    - Using testfile to test new instructions.

Result

- 8. Verify the result.

```
# Your lw instruction is correct!
# Your lw instruction is correct!
# Your add instruction is correct!
# Your sub instruction is correct!
# Your and instruction is correct!
# Your beq/or instruction is correct!
# Your slt instruction is correct!
# Your sw/lw instruction is correct!
# Your jal/add instruction is correct!
# Your beq/add instruction is correct!
# Your mod instruction is correct!
#
#
#
#
# ------------------------------------------------------------
#
#   Congratulations!! Your design has passed all the test!!
#
# ------------------------------------------------------------
```

Common Workflow

- After checking instructions in hardware, now we are able to put them into software and profile it.
  - **5. Define your instruction and its functionality**
    - This step is about to define your instruction clearly, and how it works.
  - **6. Assign an unused opcode for it**
    - Opcode is the portion that specifies the operation to be performed.
  - **7. Modify the toolchain and software**
    - Build instruction into gnu-toolchain and software.
  - **8. Verify the result**
    - Execute the program and profile it.

Check(1/2)

- 12. To verify you Adding the mod Instruction to RISC-V ISA, you can try for the following C code:

```c
#include <stdio.h>
int main(){
  int a,b,c;
  a = 5;
  b = 2;
  asm volatile(
    "mod   %0, %1, %2\n\t"
    : "=r" (c)
    : "r" (a),"r" (b)
  );
  if ( c != 1 ){
      printf("\n[FAILED]\n");
      return -1;
   }
  printf("\n[PASSED]\n");
  return 0;
}
```

- 13. Compile it and see the result.

```
riscv@riscv-VirtualBox:~/riscv-code$ riscv64-unknown-elf-gcc mod.c -o mod
riscv@riscv-VirtualBox:~/riscv-code$ spike pk mod
bbl loader

[PASSED]
```

Check(2/2)

- You can also inspect the output binary file.

```
$ riscv64-unknown-elf-objdump –dC mod > mod.dump
$ vim mod.dump (or $nano mod.dump)
```

```
00000000000101b6 <main>:
   101b6:       1101                    addi    sp,sp,-32
   101b8:       ec06                    sd      ra,24(sp)
   101ba:       e822                    sd      s0,16(sp)
   101bc:       1000                    addi    s0,sp,32
   101be:       4795                    li      a5,5
   101c0:       fef42623                sw      a5,-20(s0)
   101c4:       4789                    li      a5,2
   101c6:       fef42423                sw      a5,-24(s0)
   101ca:       fec42783                lw      a5,-20(s0)
   101ce:       fe842703                lw      a4,-24(s0)
   101d2:       02e787eb                mod     a5,a5,a4
   101d6:       fef42223                sw      a5,-28(s0)
   101da:       fe442783                lw      a5,-28(s0)
   101de:       0007871b                sext.w  a4,a5
   101e2:       4785                    li      a5,1
```

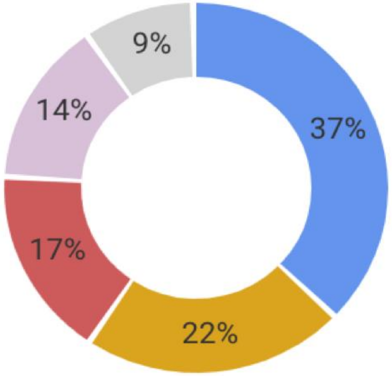
C3: Profiling

Outline

- What is Profiling
- Why Profiling
- Basic Workflow
- Example for Lab

What is Profiling

- Profiling allows you to learn where your program spent its time and which functions called which othere functions while it was executing.
- Profiler provides information that can show you which pieces of your program are slower than you expected, and might be candidates for rewriting to make your program execute faster – ***Program Optimization***



Why Profiling

- A program that hasn't been optimized will normally spend most of its CPU cycles in some particular functions.
- If we want to improve performance of our program without tools. It will take a lot of time. So we need some tools to help us to find the performance problem.
- Why we need Profiling?
  - 1. Understand our code behavior.
  - 2. Find the bottleneck of our code.
  - 3. Improve performance of our code.

Basic Workflow

- First, there are a few steps we need to know about profiling:
  - **1. Using profiler**
    Use the profiler to obtain the information that we need to optimize our program.
  - **2. Modify our program**
    Change our code according to the information provided by the profiler.
  - **3. Verify our result.**
    Confirm the program result and the execution time.
  - **4. repeat step 1 ~ step 3**
    repeat these steps until the program has optimized well enough.

C++ Profiling Example (1/2)

- In this part, we will focus on adding custom instruction to improve our performance and we will use SHA256 program as an example.
- 1. Prepare the code.(***main.cpp, sha256.h, sha256.cpp***)

```
$ mkdir sha256 && cd sha256
$ gedit main.cpp
$ gedit sha256.h
$ gedit sha256.cpp
```

  - Code Reference : ***http://www.zedwood.com/article/cpp-sha256-function***

C++ Profiling Example (2/2)

- 2. Decode gmon.out file using flat-profile mode.

```
$ riscv64-unknown-linux-gnu-gprof sha256 gmon.out -p
```

  - Result :

```
time    seconds    seconds     calls   s/call   s/call  name
36.51     12.10     12.10                                vfprintf
28.36     21.50      9.40   1000001     0.00     0.00  SHA256::transform(unsigne
unsigned int)
 9.11     24.52      3.02                                vsprintf
 3.98     25.84      1.32  64000064     0.00     0.00  SHA256_F1(unsigned int)
 3.95     27.15      1.31   1000001     0.00     0.00  sha256(std::__cxx11::basi
std::char_traits<char>, std::allocator<char> >)
 3.74     28.39      1.24  64000064     0.00     0.00  SHA256_F2(unsigned int)
 2.81     29.32      0.93                                _IO_no_init
 2.35     30.10      0.78  48000048     0.00     0.00  SHA256_F4(unsigned int)
 2.26     30.85      0.75  48000048     0.00     0.00  SHA256_F3(unsigned int)
```

- We can find out that the function "transform" have taken the most time, so we can take a look at the function first.

Add curl into SHA256

- 3. Add curl instruction and use flat-profile mode.

```
$ riscv64-unknown-linux-gnu-gprof sha256 gmon.out -p
```

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 21.89     20.04     7.02  1000001     0.00     0.00  SHA256::transform(unsign
unsigned int)
  7.67     22.50     2.46                             vsprintf
  4.58     23.97     1.47 64000064     0.00     0.00  SHA256_F1(unsigned int)
  4.49     25.41     1.44 64000064     0.00     0.00  SHA256_F2(unsigned int)
  4.15     26.74     1.33  1000001     0.00     0.00  sha256(std::__cxx11::bas
std::char_traits<char>, std::allocator<char> >)
  3.80     27.96     1.22 48000048     0.00     0.00  SHA256_F4(unsigned int)
  3.77     29.17     1.21 48000048     0.00     0.00  SHA256_F3(unsigned int)
  2.53     29.98     0.81                             _IO_no_init
  2.12     30.66     0.68                             _IO_str_init_static_inte
```

- We can find out that the self seconds of **"transform", "SHA256_F1", "SHA256_F2"** have decreased.