Program Assignment 3

Processor Speedup Strategies

Dec 05, 2018

By Ren-Shuo Liu

Contents

1 Problem

Due: 23:59 on Dec. 27, 2018

For this assignment, you're required to implement two functions, generating random numbers and searching linearly for a target number, in RISC-V assembly. Please download this ppt for detailed information.

Login to our new servers with 32-bit riscv-gcc and sodor emulator installed.

```
$ ssh -p 3456 ee3450b # or ee3450c, ee3450d
```

Add the following line to the end of **~/.tcshrc**:

```
setenv PATH $PATH\:/opt/riscv-32/bin
```

and source it:

```
$ source .tcshrc
```

Please download the sodor emulator:

```
$ cd ~/ee3450
$ git clone http://gitlab.larc-nthu.net/ee3450_2018/pa3.git
```

1.1 File Structure

```
pa3/
|— riscv-tests/
|  |— benchmarks/
|      |— Makefile
|      |— rand_and_search/
|          |— main.c       <- only edit line 22 and line 31
|          |— rand_*.S     <- implement rand() here
|          |— search_*.S   <- implement search() here
|— src/
|   |— rv32_1stage/
|       |— *.scala
|   |— rv32_5_stage/
|       |— *.scala
|— emulator/
|   |— rv32_1stage
|   |— rv32_5stage
```

The `main.c` calls different versions of `rand()` and `search()` based on the macros defined in *main.c* line 22 and line 31, respectively. For example, if `#define RAND_VER 2`, then `rand_2()`, which resides in `rand_2.S`, will be called. There are 3 versions of `rand()` and 3 versions of `search()`, defined in different assembly files, so 6 assembly files in total. Currently, all of them are empty and your job is to fill them out.

1.2 LFSR-based Random Number Generator

```
void rand(int32_t *a, int32_t len, int32_t seed);
```

The `rand()` function generates `len` pseudo-random numbers using the LFSR-based algorithm with initial value `seed`. The random numbers are stored in the integer array at address `a` in the order of their generation time. The following code snippet is a reference implementation.

```
void rand_0(int32_t *a, int32_t len, int32_t seed)
{
        a[0] = seed;
        for (int i = 1; i < len; i++) {
                int32_t tmp = a[i - 1];
                tmp = (tmp ^ (tmp << 1) ^ (tmp << 6) ^ (tmp << 7)) & 0x80;
                a[i] = (a[i - 1] >> 1) | tmp;
        }
}
```

You're required to implement the same function, but in RISC-V assembly. Further, you're required to optimize it with the new `lfsr` instruction and loop unrolling.

Different versions of rand()

RAND_VER	Function	In File	Implementation
0	rand_0	main.c	C version of rand()
1	rand_1	rand_1.S	Directly implement the rand() function in assembly
2	rand_2	rand_2.S	Based on rand_1.S, use the added LFSR instruction
3	rand_3	rand_3.S	Based on rand_2.S, unroll the loop for 4 times

Using the new LFSR instruction

To speed up random number generation, we have implemented an LFSR hardware in the ALU for you. The new ALU can not only add two numbers, shift a number, etc., it can also generate a random number directly. The new LFSR instruction is a R-type instruction, in the form of `lfsr rd, rs1, rs2`,

where `rd` stores the generated number, `rs1`stores the previously generated number (for the first generated number, `rs1` should store `seed`), and rs2 is constantly fixed to 0x380000c3.
However, as we do not modify the assembler, you can't directly call the LFSR instruction like: `lfsr t1, a2, t0`. Instead, you need to know the instruction format and write the LFSR instruction in raw bits.
For example, say you want to use `lfsr t1, a2, t0`, you should write `.word 0x56730b`, since:

```
func7 = 0 (fixed)

rs2 = 5 (t0)

rs1 = 12 (a2)

func3 = 7 (fixed)

rd  = 6 (t1)

opcode = 11 (fixed)
```

and the format of a RISC-V R-type instruction is:

```
| func7 (7 bits) | rs2 (5 bits) | rs1 (5 bits) | func3 (3 bits) | rd (5 bits)
 | opcode (7 bits) |
```

The id of each register can be found here. Thus, the final bit pattern is

```
0000000 00101 01100 111 00110 0001011

0       5     12    7   6     11
```

Grouping four bits into one chunk:

```
0000 0000 0101 0110 0111 0011 0000 1011

0    0    5    6    7    3    0    b
```

we obtain `.word 0x56730b`.

1.3 Linear Search

```
int32_t search(int32_t *a, int32_t len, int32_t target);
```

The `search()` function searches for the number `target` in an integer array linearly and return the index of `target`upon a successful query; otherwise, return -1. The integer array is at address `a` and has `len` elements. The following code snippet is a reference implementation.

```c
int32_t search_0(int32_t *a, int32_t len, int32_t target)
{
        for (int32_t i = 0; i < len; i++)
                if (a[i] == target)
                        return i;
        return -1;
}
```

You're required to implement the same function, but in RISC-V assembly. Further, you're required to optimize it with loop unrolling and static instructions reordering.

Different versions of search()

SEARCH_VER	Function	In File	Implementation
0	search_0	main.c	C version of search()
1	search_1	search_1.S	Directly implement the search() function in assembly
2	search_2	search_2.S	Based on search_1.S, unroll the loop for 4 times (but don't reorder the load instructions with other instructions)
3	search_3	search_3.S	Based on search_2.S, reorder the load instructions with other instruction to avoid data hazards

1.4 Compile and Run

Compile your **rand and search** program,

```
$ cd ~/ee3450/pa3/riscv-tests/benchmarks
$ make
```

Run your program on the 1-stage and 5-stage emulators,

```
$ make run-1stage

$ make run-5stage
```

If your program is correct, the cycle counts of `rand()` and `search()` will be displayed on the terminal.

1.5 Grading

Grading is based on the performance of your **rand and search** program running on the 1-stage and 5-stage emulators. We will only measure the 3rd version of `rand()` and `search()`, so make sure your submission includes rand_3.S and search_3.S.

1-stage emulator

rand() - 25%

	Cycle Count Range	Score
Tier A	x <= 900	100
Tier B	900 < x <= 1400	90
Tier C	1400 < x <= 2100	80
Tier D	2100 < x <= 2800	70
Tier E	x > 2800	60

search() - 25%

	Cycle Count Range	Score
Tier A	x <= 3200	100
Tier B	3200 < x <= 3400	90
Tier C	3400 < x <= 3600	80
Tier D	3600 < x <= 3800	70
Tier E	x > 3800	60

5-stage emulator

rand() - 25%

	Cycle Count Range	Score
Tier A	x <= 1000	100
Tier B	1000 < x <= 1700	90
Tier C	1700 < x <= 2600	80
Tier D	2600 < x <= 3500	70
Tier E	x > 3500	60

search() - 25%

	Cycle Count Range	Score
Tier A	x <= 3900	100
Tier B	3900 < x <= 4200	90
Tier C	4200 < x <= 4800	80
Tier D	4800 < x <= 5300	70
Tier E	x > 5300	60

1.6 Submission

Please zip the rand_and_search directory, name your zip file studentid.zip, and submit your code via the link.

```
$ cd ~/ee3450/pa3/riscv-tests/benchmarks
$ tar -cvf [your student id].tar rand_and_search
```

2 Issues

If you encounter any server error or do not understand the problem description, please contact:

Yun-Sheng Chang <yschang@gapp.nthu.edu.tw>