# Computer Architecture



# Program Assignment 2

May 31, 2020 (2020-05-31T00:00:00+08:00)

By Ren-Shuo Liu

**Due: 23:59 on May. 31, 2020**

Contents



# 1 Problem

**Due: 23:59 on May. 31, 2020**


## Pages

- Contacts
- Course Video
- Important Dates
- Notes
- Resources
- Rules

## Categories

labs

misc

programming assignments

tutorials

## Social

Q & A Forum

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 (./downloads/sodor.pptx) for detailed information.

First, please download the sodor emulator,

```
$ ssh -p 3450 ee3450b
$ git clone http://gitlab.larc-nthu.net/ee3450/sodor.git pa2
```

## 1.1 File Structure

```
pa4/
|— 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 (https://en.wikipedia.org/wiki/Linear-feedback_shift_register)-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 |

## Use the new LFSR instruction

To speed up random number generation, we implement a LFSR hardware in the CPU emulator. I.e., the ALU now 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 always fixed to 0x380000c3.

However, since we do not modify the assembler, you can't directly use the LFSR instruction as a typical instruction, e.g. `lfsr t1, a2, t0`. Instead, you should recognize the bit pattern of the instruction and write down the bit pattern. 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) |
```

For example, say you intend to call `lfsr t1, a2, t0`, you should write `.word 0x56730b` instead, since

```
func7 = 0 (fixed)
rs2 = 5 (t0)
rs1 = 12 (a2)
func3 = 7 (fixed)
rd  = 6 (t1)
opcode = 11 (fixed)
```

The id of each register can be found here (http://www1.ee.nthu.edu.tw/ee345000/lab-1-assembly-language-and-isa-simulators.html#introduction-of-risc-v-instruction-set). The final bit pattern is

```
0000000 00101 01100 111 00110 0001011
0       5     12    7   6     11
```

Group four bits into one chunk

```
0000 0000 0101 0110 0111 0011 0000 1011
0    0    5    6    7    3    0    b
```

# 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.

```
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

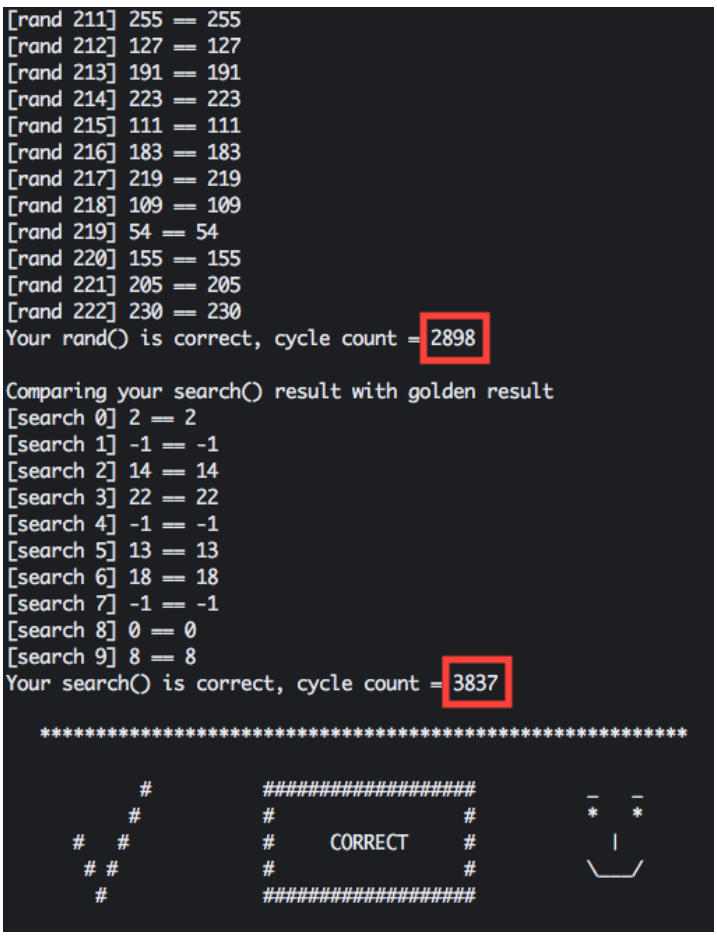
Compile your **rand and search** program,

```
$ cd ~/ee3450/pa4/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

1. Please put rand_3.S and search_3.S in the same directory named code.

```
code/
|── rand_3.S
|── search_3.S
```

2. Tar the directory and named as code.tar.

```
$ tar cvf code.tar code
```

3. Submit your code.tar to autolab (https://autolab.larc-nthu.net/courses/ee3450_2020_spring/assessments/pa2).

# 2 Issues

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

Jun Shen Wu <sesshoumaru99912@gmail.com (mailto:sesshoumaru99912@gmail.com)> Yun Chen Lo <tarzen0509@gmail.com (mailto:tarzen0509@gmail.com)>

---