Checkpoints specifications

1. Dump memory and register

This section specifies the usage of checkpoints.txt file. The file specifies the locations which you need to dump 2 files under the current directory right after executing of the instructions. The number in checkpoint.txt refers to the **number of instructions you have simulated(instruction count)**.

`memory_x.bin:` The contents of the 6MB memory you simulated at the current state. The file should have the size of 6MB.

`register_x.bin:` The contents of registers, which in total should have the size of 32*4B.

Here is an example for dumping. Assume that we have a part of assembly codes from *fib.asm*:

```
    .text
    addi $v0, $zero, 5 # 1st instruction
    syscall
    add $s1, $zero, $v0


    lui $at, 80 # 4th instruction
    ori $a0, $at, 0
    addi $v0, $zero, 4
    syscall

    addu $a0, $s1, $zero # 8th instruction
    ...
```

And *fib_checkpts.txt* will have several lines. Each line will contain a number, in increasing order:

```
0
3
7
```

In this case, it means that after executing the No.0(to be clear, 0 means after initialization and right before the first instruction), 3rd and 7th instruction of the machine code, you should dump two binary files (memory_x.bin, register_x.bin where x $\in$ *{0,3,7}*)to show your process of running the program. Hence after the program finishes execution, the following snapshots should be under the same directory as the simulator:

```
memory_0.bin
register_0.bin
memory_3.bin
register_3.bin
```

```
memory_7.bin
register_7.bin
```

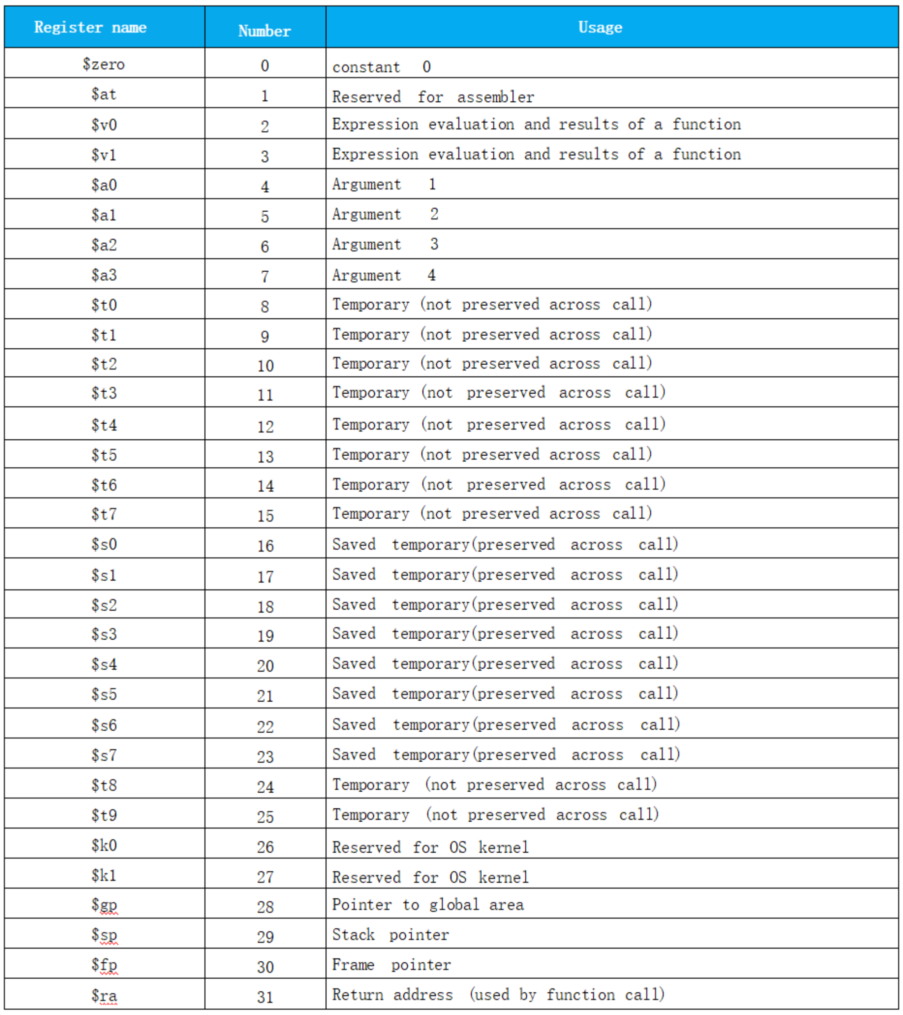
Note that the purpose of dumping is to check if your simulator has some correct parts during the process of running, so that you may still get some points even if you failed a test. You can check the correctness of your intermediate files using the following commands in linux terminal:

```
cmp memory_x.bin memory_correct_x.bin
cmp register_x.bin register_correct_x.bin
```

Please remember to dump the registers in order.

Since we need to check the correctness of the register dump, you need to dump registers in a certain order:

1. dump the 32 general purpose registers by the order in the following picture
2. dump the PC register, the HI register, and the LO register one by one



Register name	Number	Usage
$zero	0	constant 0
$at	1	Reserved for assembler
$v0	2	Expression evaluation and results of a function
$v1	3	Expression evaluation and results of a function
$a0	4	Argument 1
$a1	5	Argument 2
$a2	6	Argument 3
$a3	7	Argument 4
$t0	8	Temporary (not preserved across call)
$t1	9	Temporary (not preserved across call)
$t2	10	Temporary (not preserved across call)
$t3	11	Temporary (not preserved across call)
$t4	12	Temporary (not preserved across call)
$t5	13	Temporary (not preserved across call)
$t6	14	Temporary (not preserved across call)
$t7	15	Temporary (not preserved across call)
$s0	16	Saved temporary(preserved across call)
$s1	17	Saved temporary(preserved across call)
$s2	18	Saved temporary(preserved across call)
$s3	19	Saved temporary(preserved across call)
$s4	20	Saved temporary(preserved across call)
$s5	21	Saved temporary(preserved across call)
$s6	22	Saved temporary(preserved across call)
$s7	23	Saved temporary(preserved across call)
$t8	24	Temporary (not preserved across call)
$t9	25	Temporary (not preserved across call)
$k0	26	Reserved for OS kernel
$k1	27	Reserved for OS kernel
$gp	28	Pointer to global area
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address (used by function call)

2. Example code of dump memory

Since this part of work is only to increase the accuracy of scoring, we don't want to burden you.Therefore, we give out an example implementation of the dumping process.

To be clear, you don't have to use these codes. You can implement the dumping process in your own way.

```
#include <set>
#include <cstdio>

// checkpoint
std::set<int> checkpoints;
void init_checkpoints(char* checkpoint_file) {
  FILE *fp = fopen(checkpoint_file, "r");
  int tmp, i = 0;
  while(fscanf(fp, "%d", &tmp) != EOF){
        checkpoints.insert(tmp);
  }
}

void checkpoint_memory(int ins_count) {
  if (!checkpoints.count(ins_count))
    return;
  std::string name = "memory_" + std::to_string(ins_count) + ".bin";
  FILE * fp = fopen(name.c_str(), "wb");
  fwrite(bs, 1, 0x600000, fp);
  fclose(fp);
}

void checkpoint_register(int ins_count) {
  if (!checkpoints.count(ins_count))
    return;
  std::string name = "register_" + std::to_string(ins_count) + ".bin";
  FILE * fp = fopen(name.c_str(), "wb");

  fwrite($zero, 4, 1, fp);
  fwrite($at, 4, 1, fp);
  fwrite($v0, 4, 1, fp);
  fwrite($v1, 4, 1, fp);
  fwrite($a0, 4, 1, fp);
  fwrite($a1, 4, 1, fp);
  fwrite($a2, 4, 1, fp);
  fwrite($a3, 4, 1, fp);
  fwrite($t0, 4, 1, fp);
  fwrite($t1, 4, 1, fp);
  fwrite($t2, 4, 1, fp);
  fwrite($t3, 4, 1, fp);
  fwrite($t4, 4, 1, fp);
  fwrite($t5, 4, 1, fp);
  fwrite($t6, 4, 1, fp);
  fwrite($t7, 4, 1, fp);
  fwrite($s0, 4, 1, fp);
  fwrite($s1, 4, 1, fp);
  fwrite($s2, 4, 1, fp);
  fwrite($s3, 4, 1, fp);
  fwrite($s4, 4, 1, fp);
  fwrite($s5, 4, 1, fp);
  fwrite($s6, 4, 1, fp);
```

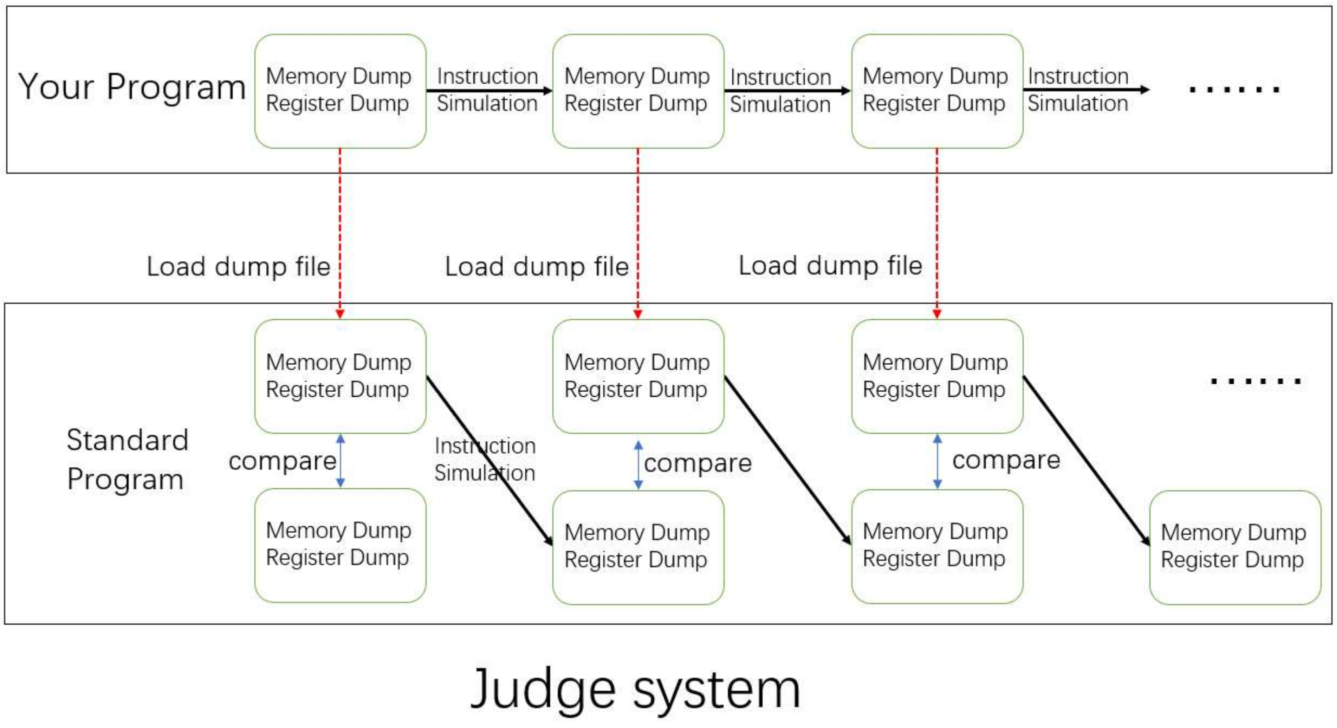
```
    fwrite($s7, 4, 1, fp);
    fwrite($t8, 4, 1, fp);
    fwrite($t9, 4, 1, fp);
    fwrite($k0, 4, 1, fp);
    fwrite($k1, 4, 1, fp);
    fwrite($gp, 4, 1, fp);
    fwrite($sp, 4, 1, fp);
    fwrite($fp, 4, 1, fp);
    fwrite($ra, 4, 1, fp);

    fwrite($pc, 4, 1, fp);
    fwrite($hi, 4, 1, fp);
    fwrite($lo, 4, 1, fp);

    fclose(fp);
  }
```

3. Grading process

There will be several test cases. Ideally, we try to grade your program by the number of correct-simulated instructions. The grading process are shown as follows:



However, even we try to eliminate the influence of your cumulated errors by loading your pre-memory-dum p fi les before each instruction simulated, there are some exceptional cases that may break off your simulation, such as dividing by zero or other segment faults. We will increase the number of test cases to mitigate their influence.