Data Structrue:

Class Cache:

1. To simulate Cache behavior, read .org file to initialize the properties of the cache, such as: address bits, block_size...Then, following the requirement, use NRU as replacement policy. Implement NRU with a reference array. There is 1 reference bit for each entry. To run benchmark, implement a function run_bench to access every address in the benchmark. Fetch each address and hit the cache if the data has been already in the cache. If not execute replacement policy. it will calculate the miss time.

2. Variables in Class:
   (1) indexing_bits: An vector that record the position of indexing bits
   (2) cache_entry: An vector that record the addreess of data in cache
   (3) valid_bits: An vector that record the valid reference bit of entries of cache
   (4) nru_table: An vector that record the reference bit of the implement of NRU policy

```
    int address_bits;
    int block_size;
    int cache_sets;
    int associativity;
    int cache_size;
    bool is_setup;

    int offset_bit_count;
    int indexing_bit_count;
    std::vector<int> indexing_bits;
    std::vector<int> cache_entry;
    std::vector<bool> valid_bits;
    std::vector<bool> nru_table;3.
```

3. Functions
   (1) Constructor:

```
Cache(std::fstream *org_file)
```

   (2) Function setup:

   (3) NRU policy:
   To modulize the program, implement NRU policy in function replace and nru_hit_policy individualy. Function replace can set and change the reference bit and replace the data in cache.

```
bool replace(std::vector<bool> target_vb)// NRU Policy
```

   Function nru_hit_policy can call function hit whether the required data is in cache or not and change the reference bit to 1.

```
bool nru_hit_policy(int idx)
```

(4) Function hit:
It would check the required data is in cache or not.

```
bool hit(std::vector<bool> target_vb){
```

(5) Function fetch:
Call function hit to determine wheher the data of target_address in cache or not. If return miss, execute function replace.

```
bool fetch(std::vector<bool> target_address){
```

(6) Function output_cache_info:
Output the property of cache as the final project manual required to .rpt file.

```
bool output_cache_info(std::fstream *file){
```

(7) Function run_bench:
Run and test the input benchmark and output result to .rpt file. It would execute fetch(taget_address) for each address in benchmark. The function will return hit rate.

```
float run_bench(Bench b, std::fstream *file, bool
is_output){
```

(8) Function show_cache:

```
bool show_cache(){
```

(9) Function clear_cache:
Clear all data of cache_entry, valid_bits, nru_table. Reset the data in cache.

```
bool clear_cache(){
        this->cache_entry.clear();
        this->valid_bits.clear();
        this->nru_table.clear();
    }
```

(10) Fucntion clear_all:
Clear all properties that already set.

```
    bool clear_all(){
```

Structure Bench:
The data structue of benchmark.

```
typedef struct bench{
    std::string bench_name;
    std::vector<std::vector<bool>> bench_data;
}Bench;
```

Structure bit_set:
The data structure uesd in selecting best index bits. It record the bit pair, target address of benchmark and correlation score of the pair.

```
typedef struct bit_set{
    std::vector<int> bits;
    std::vector<int> samples;
    float score;
}Bit_Set;
```

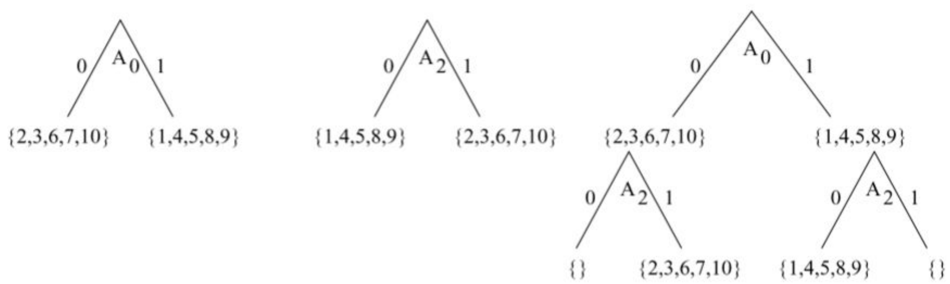
Implement of Heuristic Method
It is a implement of the reference: <<Zero Cost Indexing for Improved Processor Cache Performance>>. Select the indexing bits that distribute the target address of benchmark most uniformly.
To measure how good the selected index is. The article develop a quality score to to it. It compare address number 1,0 of the selected bit. If “0” is more than “1”, Q = number of “1” / number of “0”. Vice versa. If “1” is more than “0”, Q = number of “0” / number of “1”.

$$Q_i = \frac{\min(Z_i, O_i)}{\max(Z_i, O_i)}. \tag{5}$$

In Eq. (5), Z_i denotes the number of references having the value zero at address bit A_i and O_i denotes the number of references having the value one

at address bit A_i. We further illustrated the concept of quality measure with a

To measure how good a pair of selected indeice are. We can image a tree that if select address bits A1 and A2, address bit A1 would divide the address into “1” and “0” two group, same as A2. Then with A1, A2, it will divide address into 4 group. The best way to divide the address is make the address belong to 4 group uniformly.



After that, we can develop a policy to measure the correlation score of selected bits. E denote the number of reference having identical values between A1, A2. D denote the number of reference having different values between A1, A2.

measure $C_{i,j}$ equal to zero. In general, we can compute the correlation $C_{i,j}$, for address bits A_i and A_j as shown in Eq. (7).

$$C_{i,j} = \frac{\min(E_{i,j}, D_{i,j})}{\max(E_{i,j}, D_{i,j})}. \tag{7}$$

In Eq. (7), $E_{i,j}$ denotes the number of references having identical values at address bits A_i and A_j. Likewise, $D_{i,j}$ denotes the number of references having different values at address bits A_i and A_j. Applying Eq. (7) to our running example, we compute the correlation measures shown in Table IV.

In the program, use a special data structure "Bit_set" to record the selected indice, score of index bits, and the address equal to "1" in attribute sample.

Functions:

(1) Function quality_score:
Calculate quality score mentioned above and return the score.

```
float quality_score(int bench_size, int true_times){
```

(2) Function correlation_score:
Calculate correlation score mentioned above and return the score.

```
float correlation_score(int a_sample_size, int b_smaple_size, int
inter_sample_size){
```

(3) Funciton intersect:
Intersect 2 pairs of bit_set. It would return a new Bit_Set data of bit_set a and b. The indice of a, b would be unioned. The sample of returned Bit_Set is the intersection of bit_set a, b. It will also call function correlation to compute the correlation score of returned bit_set.

```
Bit_Set intersect(Bit_Set a, Bit_Set b){
```

(4) Function best_bit_set:
Find the best pair of bits via the method above. It calculate the correlation score of all the possible pair of 2 bits. Then, to reduce the number of the combinations of bits pairs, it would sort all computed pairs and select best N pairs remain to next iteration. In the next iteration, it will compute 3 bits and then 4 bits… until it reach the required counts of bits.

```
Bit_Set best_bit_set(std::vector<Bit_Set> indexing_bit_set, int
indexing_bit_count){
```

(5) Function select_indexing_bits
To find the best indexing bit by the method above, call this function. It would initialize the every bit with a Bit_set data streucture. It would compute the score and samples of the index bit. Finally, it executes function best_bit_set to compute the possible combinations of pairs of bits.

```
std::vector<int> select_indexing_bits(Bench bch, Cache c){
```

Utilities Function and Variables

Variables

Variable debug_mode represent whether to show data in terminal.

```
const bool debug_mode = false;
const bool sel_idx_debug = false;
```

Functions

Function vbtoi:

Convert binary index array to integer.

```
int vbtoi(std::vector<bool> index){
```

Function vbtos:

Convert binary index array to string.

```
std::string vbtos(std::vector<bool> index){
```

Function trim_offset:

Trim logN offset bits of address. N = bock size

```
std::vector<bool> trim_offset(std::vector<bool> address, int
offset_bit_count){
```

Function select_address:

Select the bits of address in the position of indexing_bits.

```
std::vector<bool> select_address(std::vector<bool> address,
std::vector<int> indexing_bits){
```

Function read_stream:

Read data from spacific file.

```
std::string read_stream(std::fstream *file, bool is_show){
```

Function

Write data to spacific file.

```
bool write_file(std::fstream *file, std::string str, bool
is_newline, bool is_show){
```

Function

Write data report to spacific file.

```
bool write_log(std::string str, bool debug_mode){
```

Function

Read benchmark data from spacific file.

```
Bench read_bench(std::fstream *file){
```

Function

Show benchmark data from spacific file.

```
bool show_bench(Bench bch){
```

Flow Chart: