HW6

How to submit:

Save all your answers to hw6.txt and type the following to submit after you finish all the problems:

~lyang11/bin/submit cs304 hw6 hw6.txt

1. Structure and alignment

Consider the following C declaration:

```
struct Node{
     int value;
     char c;
     struct Node* left;
     char d;
     short flag;
     struct Node* right;
     short unit;
     struct Node* next;
  };
typedef struct Node* pNode;
/* NodeTree is an array of N pointers to Node structs */
#define N 100
pNode NodeTree[N];
```

In this problem, the program runs on X86-64.

Using the template below (allowing a maximum of 64 bytes), indicate the allocation of data for a **Node** struct. Mark off and label the areas for each individual element (there are 8 elements in the struct). Cross hatch the parts that are allocated but not used (to satisfy alignment).

Assume the Linux alignment rules discussed in class. **Clearly indicate the right hand boundary of the data structure with a vertical line**.

First 32 bytes (starting from lower address):

```
||---+---+---+---+---+---+---+---||---+---+---+---+---+---+---+---||---+---+---+---+---+---+---+---||---+---+---+-
--+---+---+---+---||
```

Second 32 bytes:

```
||---+---+---+---+---+---+---+---||---+---+---+---+---+---+---+---||---+---+---+---+---+---+---+---||---+---+---+-
--+---+---+---+---||
```

For each of the three following C references, please indicate which assembly code section (labeled A through F below) places the value of that C
reference into register %rax. If no match is found, please write ``NONE" next to the C reference.

The register-to-variable mapping for each assembly code section is:

%rdi = starting address of the NodeTree array
%esi = i (initially)
%rax = result

C References:

1. ______ NodeTree[i]->right->left->next

You can first write down the address of NodeTree[i], and then NodeTree[i]->right, then NodeTree[i]->right->left, and then NodeTree[i]->right->left->next.

2. ______ NodeTree[i]->left->right->next

3. ______ NodeTree[i]->next->right->left

Linux x86-64 Assembly fragments:

A.
```
      movslq  %esi, %rsi
      movq    (%rdi,%rsi,8), %rax
      movq    40(%rax), %rax
      movq    8(%rax), %rax
      movq    24(%rax), %rax
```

B.
```
      movslq  %esi, %rsi
      movq    (%rdi,%rsi,8), %rax
      movq    24(%rax), %rax
      movq    8(%rax), %rax
      movq    40(%rax), %rax
```

C.
```
      movslq  %esi, %rsi
      movq    (%rdi,%rsi,8), %rax
      movq    24(%rax), %rax
      movq    40(%rax), %rax
      movq    8(%rax), %rax
```

D.
```
      movslq  %esi, %rsi
      movq    (%rdi,%rsi,8), %rax
      movq    8(%rax), %rax
      movq    24(%rax), %rax
      movq    40(%rax), %rax
```

E.
```
      movslq  %esi, %rsi
      movq    (%rdi,%rsi,8), %rax
      movq    8(%rax), %rax
      movq    40(%rax), %rax
```

```
        movq    24(%rax), %rax
```

F.

```
        movslq  %esi, %rsi
        movq    (%rdi,%rsi,8), %rax
        movq    40(%rax), %rax
        movq    24(%rax), %rax
        movq    8(%rax), %rax
```

2. Union

In the space provided below, give the output of the C program shown. For information about strcpy, see http://icecube.wisc.edu/~dglo/c_class/strmemfunc.html or type man strcpy. The ASCII values of the characters A through E are 0x41 through 0x45, respectively. Recall that the format string %08x specifies that the output item with which it matches should be printed in a field of at least 8 hexadecimal characters wide with leading zeros, if required. The other format strings have similar definitions (see here http://icecube.wisc.edu/~dglo/c_class/printf.html). For this example, you notice that the starting address of yew.i[0], yew.s[0], and yew.c[0] are the same. Suppose this program runs on x86 machine.

```
union {
  int   i[2];
  short s[3];
  char  c[6];
} yew;

int main()
{
  yew.i[0] = 0x12345678;
  yew.i[1] = 0xfedca987;
  printf("%04hx %04hx %04hx\n",yew.s[0],yew.s[1],yew.s[2]);
  strcpy(yew.c,"ABCDE");
  printf("%08x %08x\n",yew.i[0],yew.i[1]);
  yew.s[0]=0xa1b2;
  yew.s[1]=0xc3d4;
  yew.s[2]=0xe5f6;
  yew.s[3]=0x1728;
  printf("%08x %08x\n",yew.i[0],yew.i[1]);
  return 0;
}
```

OUTPUT:

3. Set associative cache

Suppose a computer's address size is m bits (using byte addressing), the cache size is C bytes, the block size is B bytes, and the cache is E-way set-associative. Assume that B is a power of two, so B = 2^b. Figure out what the following quantities are in terms of C, B, E, b, and m: the number of sets in the cache, the number of index bits in the address, and the number of bits for a tag.

4. Cache dimensions

a. How many 32-bit integers can be stored in a byte-addressed direct-mapped cache with 15 tag bits, 15 set index bits, and 2 block offset bits? Write your answer as a power of 2.
b. Assume 16-bit byte addresses. You are trying to access memory byte address 0x3434. What are the corresponding tag, index and offset for
   i. 32-line direct-mapped cache with 4 byte blocks?
   ii. 16-line direct-mapped cache with 16 byte blocks?
   iii. 1KB direct-mapped cache with 8 byte blocks?
c. What is the ratio of data bits to total bits in a 128 byte write-back direct-mapped cache that has 8-byte blocks and byte addresses are 64 bits? What about if it were a write-through cache instead? Data bits means the bits for real data (not including tag bits, valid bits, or dirty bits). **Note:** In write-back cache, only when a cache block is replaced, if the data in the cache block was modified, the data will be written back to the memory. In this case, we need a drity bit to show that the data has been modified. If the dirty bit is 0, the data do not have to written back to the memory. In write-through cache, when a cache block is written, the data is also written to the memory at the same time. So we do not need a dirty bit.

5. Cache accesses

The following C program is run (with no optimizations) on a processor with a direct-mapped cache that has 32 byte blocks and holds 1024 bytes of data:

```
int i,j,c,stride,array[1024];
...
for(i=0; i<10000; i++)
  for (j=0; j<1024; j+=stride)
        c += i%(array[j]);
```

If we consider only the cache activity generated by references to the array and we assume that integers are four bytes, what are possible miss rates (there is more than one for one of them) when

i. stride = 256?
ii. stride = 255?

Explain your answers clearly in a few sentences. Stride is defined in the program. It is the index difference between two consecutive accesses to the array. In this problem, you can assume the cache only contains the data for array[]. An easy way is to see where you will place each integer in the cache instead of using the TIO to calculate the index.

For this problem, you should assume the starting address of array[1024] (i.e., array) can be anywhere in memory as long as array%4==0 (for alignment purpose). Remember that array[1024] occupies 1024*4 Bytes of memory. The address of array[255] (or &(array[255])) is actually (char *)array+255*4.

6. Average memory access time

L1 Cache has a hit rate of 80% and a hit time of 1 cycle. L2 Cache has a local miss rate of 10% and a hit time of 4 cycles. L3 Cache has a local miss rate 5%, and a hit time of 20 cycles. Accessing main memory takes 100 cycles. Find the AMAT (average memory access time).

In the problem, you should first calculate the AMAT for L3 cache, and then L2, and L1. L2 has a local miss rate of 10% means that, among all the data accesses that result in cache misses in L1 cache, 10% of those will encounter cache misses in L2 cache.

In this case, L3 AMAT= L3 hit time + L3 miss rate * L3 miss penalty. And this L3 AMAT is the miss penalty of L2 cache.