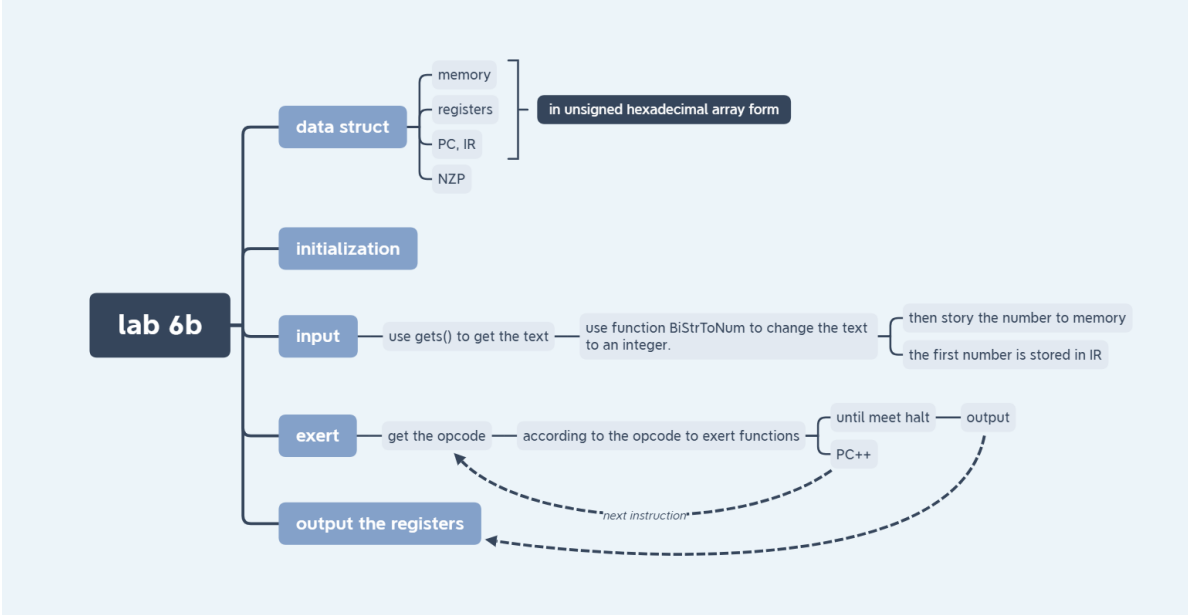
lab 6 report



1.data struct

Because the short int data type has exactly 16 bits, I use unsigned short int form to store the information.

unsigned short int is used as instructions which stored in memory and registers.

signed short int is used to calculate. For example, immediates and offsets are in signed short int form.

```
typedef short int int16_t;
typedef unsigned short int uint16_t;

static uint16_t mem[65536]; //memory
static uint16_t R[8];   //register
static uint16_t PC, IR;
static int N = 0, Z = 1, P = 0;
static uint16_t inst;   //instruction
```

2.main function

```
int main()
{
    //initialization
    for(int i = 0; i<8; i++)    R[i] = 0x7777;
    for(int i = 0; i<65536; i++)    mem[i] = 0x7777;

    input();
    exert();

```

```
    for(int i = 0; i < 8; i++)  //output
    {
        printf("R%d = x%04hX\n", i, R[i]);
    }
}
```

3.input

use function input() to read the program.

```
void input(void)
{
    char* line;
    uint16_t now;
    int i;

    //the first line reveals the position of the first instrucion(IR).
    line = (char*)malloc(17);
    gets(line);
    if(line != NULL)            //gets() returns NULL when meeting EOF.
    {
        IR = BiStrToNum(line);  //function BiStrToNum(char*) turns a binary string to an integer
    }
    else return;

    now = IR;                   //get the rest instructions and store them into the memory.
    PC = IR + 1;
    while(gets(line) != NULL)
    {
        mem[now++] = BiStrToNum(line);
    }
    return;
}
```

4.exert

To exert the program, we need to firstly read the opcode, and switch to different funcions according to the opcode.

```
void exert(void)
{
    int halt = 1;   //to judge whether we met the halt instruction
    while(halt)
    {
        inst = mem[IR];
        uint16_t opcode = inst >> 12;   //shift left 12 bits to get the opcode

        switch(opcode)
        {
```

```
            case BR:
                func_BR();
                break;
            case ADD:
                func_ADD();
                break;
            case LD:
                func_LD();
                break;
            case ST:
                func_ST();
                break;
            case JSR:
                func_JSR();
                break;
            case AND:
                func_AND();
                break;
            case LDR:
                func_LDR();
                break;
            case STR:
                func_STR();
                break;
            case NOT:
                func_NOT();
                break;
            case LDI:
                func_LDI();
                break;
            case STI:
                func_STI();
                break;
            case JMP:
                func_JMP();
                break;
            case LEA:
                func_LEA();
                break;
            case HALT:
                halt = 0;
                break;
        }
        IR = PC;    //end exerting, come to the next instruction.
        PC = PC + 1;
    }
}
```

For these different functions, some of them are similar or some of their codes and ideas are similar. Thus, it is meanless to put all of them in my report. I put some representative functions there.

The key thing is that how to cut out the bits we need in the instrutions. The way I find is to use a shift operation and an AND operation.

```c
void func_BR(void)
{
    int n = (inst >> 11) & 0x1; //I use this form to cut out the bits we need in the instruction
    int z = (inst >> 10) & 0x1;
    int p = (inst >> 9) & 0x1;
    int16_t offset = inst & 0x1ff;
    if((offset >> 8) > 0)       //to judge if the offset is minus
    {
        offset = offset - 0x200;
    }
    if((N && n) || (Z && z) || (P && p)) PC = PC + offset;  // if nzp satisfied, PC shifts.
    return;
}
```

```c
void func_AND(void)
{
    uint16_t DR = (inst >> 9) & 7;
    uint16_t SR1 = (inst >> 6) & 7;
    uint16_t bit5 = (inst >> 5) & 1;

    if(bit5 == 0)
    {
        uint16_t SR2 = inst & 7;
        R[DR] = R[SR1] & R[SR2];
    }
    else
    {
        int16_t imm = inst & 0x1f;
        if((imm >> 4) > 0)  imm = imm - 32;
        R[DR] = R[SR1] & imm;
    }

    nzp_update(DR); //change nzp(conditional codes) according to DR
    return;
}
```

```c
void func_JSR(void)
{
    uint16_t bit11 = (inst >> 11) & 1;

    R[7] = PC;
    if(bit11)
    {
        int16_t offset = inst & 0x7ff;  //if bit11 is 1, PC + offset
        if((offset >> 10) > 0)
        {
            offset = offset - 0x800;
        }
        PC = PC + offset;
    }
    else {                              //if bit11 is 0, PC = R[baseR]
        uint16_t baseR = (inst >> 6) & 7;
        PC = R[baseR];
    }
```

```
    return;
}
```

```
void func_LEA(void) // R[DR] = PC + offset
{
    uint16_t DR = (inst >> 9) & 7;
    int16_t offset = inst & 0x1ff;
    if((offset >> 8) > 0)
    {
        offset = offset - 0x200;
    }
    R[DR] = PC + offset;
    return;
}
```

To change the conditional codes, I write the function nzp_update to update them. It is easy to comprehense.

```
void nzp_update(uint16_t DR)
{
    if(R[DR] == 0)
    {
        N = 0;  Z = 1;  P = 0;
    }
    else if((R[DR] >> 15) > 0)
    {
        N = 1;  Z = 0;  P = 0;
    }
    else
    {
        N = 0;  Z = 0;  P = 1;
    }
    return;
}
```

5.output

It is easy to output all the registers. I put these codes in the main function.

```
for(int i = 0; i < 8; i++)  //output
    {
        printf("R%d = x%04hX\n", i, R[i]);
    }
```