《程序设计课程设计》实验报告

实验名称　　《模拟 CPU 程序设计》概要设计

班　　级　　**2019211318 班**

姓　　名　　叶淳瑜

学　　号　　**2019212686**

1 高层数据结构设计

全局常量/变量定义

为了追求高内聚低耦合的性质，该文件尽量少的使用全局变量。

```
typedef struct Instrution            //代码链表结点
{
    int op;                          //操作码
    int rs;                          //源操作数寄存器 1
    int rt;                          //源操作数寄存器 2
    int im;                          //立即数
    int address;                     //指令地址
    int dec_instruction_high;        //十进制高 16 位指令
    char bin_instruction[34];        //二进制指令
    int id;                          //线程 id
    struct Instrution *next;
} * LinkInstruction, Instruction;    //代码链表结点

#define DATAMAX 1024
#define CODEMAX 512
#define TICKETS 100          //售票总数
#define CODEBEGIN_1 0        //代码表 1 内存起始位置
#define CODEBEGIN_2 256 //代码表 2 内存起始位置

int pc[2] = {0}, ir[2] = {0}, fr[2] = {0};          //系统寄存器
int gr[2][9] = {0};                                 //通用寄存器
int dataSegment[DATAMAX] = {0};                     //数据段(每个数代表 2Byte)
int codeSegment[CODEMAX] = {0};                     //代码段(每个数代表 4Byte)
HANDLE hMutex1, hMutex2;                            //互斥对象句柄
```

注：CPU 寄存器设计采用数组，下标为 0 为线程 1 的寄存器，下标为 1 为线程 2 的寄存器

各模块常量与变量定义

main.c:

LinkInstruction code1, code2; //代码链表头结点

FILE *codeFile1, *codeFile2; //文件

HANDLE thread1, thread2; //线程 1，2 句柄

loadCode.c:

int weight; //二进制位权

int dec; //10 进制数

char tmp[34]; //2 进制指令字符串

runCode.c:

LinkInstruction cur; //代码链表指针

LinkInstruction cache; //高速缓存器

analyse.c:

int offset; //地址偏移量

int i; //寄存器数组下标

2 系统模块划分

2.1 系统模块结构图

模块划分思路说明

模块分为 main.c, loadCode.c, analyse.c, runCode.c, definition.h，分别对应了主函数，载入指令，分析指令，执行指令模块，头文件，模拟了冯·诺依曼体系结构 CPU 的真实执行过程。

其中，analyse.c, runCode.c 模块需要独立线程（以用红色标记）。在 main.c 模块中，调用 _beginthreadex 先后启用线程 1，线程 2，然后等待指令分析、执行，使用 WaitForSingleObject 等待线程结束，回到 main.c 模块。

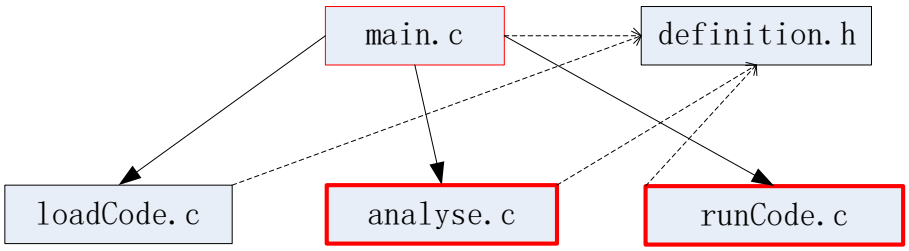


图 2-1 模块结构图

1. **模块名称 main.c**

模块功能简要描述：主函数，连接各个功能函数，完成主要功能

2. **模块名称 loadCode.c**

模块功能简要描述：载入代码，将代码段存入代码链表之中

3. **模块名称 analyse.c**

模块功能简要描述：分析指令类型，并跳转到相应函数

4. **模块名称 runCode.c**

模块功能简要描述：控制指令多线程执行，输出对应寄存器状态以及内存状态

5. **模块名称 definition.h**

模块功能简要描述：头文件，包含程序使用的标准头文件，所有函数的定义，结构体的定义等

2.2 各模块函数说明

序号	函数原型	功能	参数	返回值
1	int main(void);	主函数，协调整个 CPU 功能	void	0
2	void loadCode(LinkInstruction code, FILE *fp, int begin);	将指令载入到代码链表中	代码表，文件指针，代码表在内存中的起始位置	void

3	unsigned __stdcall runCode(void *ptr);	线程函数，分析并执行指令	代码表结构体	unsigned __stdcall 函数约定调用
4	int B2D_trasition(char *s, int begin, int end, int mode);	2 进制转 10 进制(包括原码与反码)	2 进制串，起始位置，终止位置，模式	2 进制数对应的 10 进制整数
5	void printStatus(Instruction cache);	输出各寄存器状态	高速缓存器 cache	void
6	void printMemory();	输出内存状态	无	void
7	void stop(Instruction cache);	停机指令	cache	void
8	void mov(Instruction cache);	数据传送指令	cache	void
9	void cmp(Instruction cache);	比较指令	cache	void
10	void jmp(Instruction cache);	跳转指令	cache	void
11	void arithmetic_operation(Instruction cache);	算术运算指令	cache	void
12	void logic_operation(Instruction cache);	逻辑运算指令	cache	void
13	void standard_io(Instruction cache);	输入输出指令	cache	void
14	void para_exe(Instruction cache);	控制内存访问，控制 CPU 休眠	cache	void
15	LinkInstruction initInstruction();	初始化链表结点	无	指向代码结点的指针

2.3 函数调用图示及说明

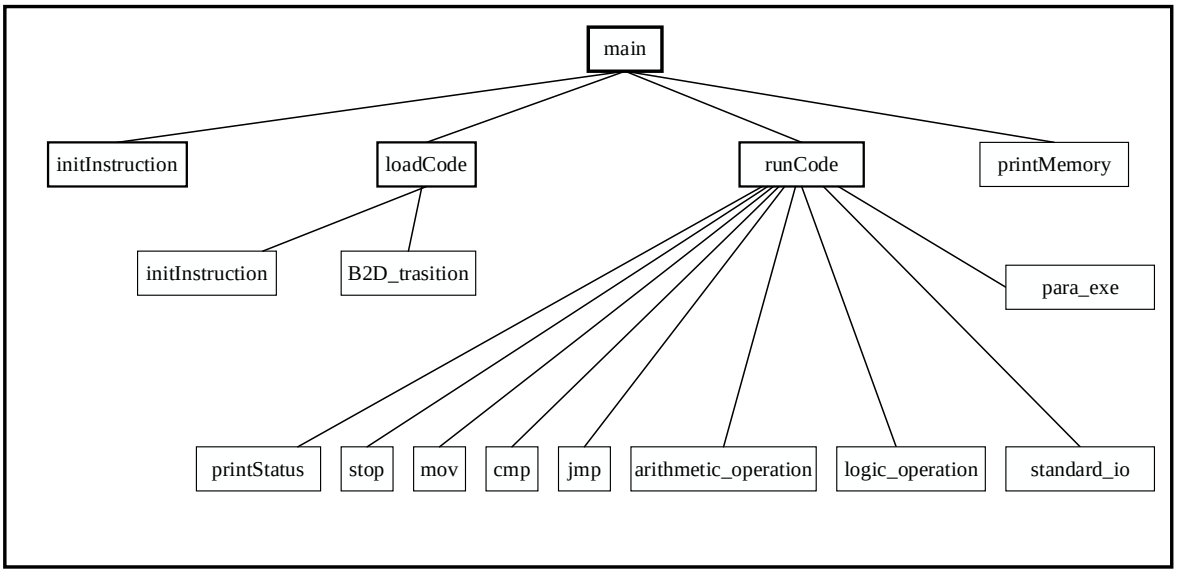


图 2-2 函数调用关系图

解释说明

1. main 函数通过调用函数 initInstruction，完成代码链表头结点的初始化。
2. main 函数通过调用函数 loadCode，线程函数 runCode，完成载入指令、多线程分析指令与执行指令。
3. main 函数通过调用函数 printMemory，输出内存状态
4. loadCode 函数通过对函数 initInstruction 与 B2D_trasition 的调用，完成代码链表的初始化与完成 2 进制、10 进制之间的转换并写入到代码链表中。
5. runCode 函数调用 printStatus，完成寄存器状态的输出。
6. runCode 函数通过调用函数：stop, mov, cmp, jmp, arithmetic_operation, logic_operation, standard_io, para_exe 等函数，完成对各条指令的分析与执行。

3 高层算法设计

主要步骤

程序的主要步骤为，对指令集文件中的每一行由 32 位二进制数字组成的指令，进行逐个读取，并存入内存的代码链表之中，并设置线程 id，链表结点包括以下内容：

```
typedef struct Instrution
{
    int op;                     //操作码
    int rs;                     //源操作数寄存器1
    int rt;                     //源操作数寄存器2
    int im;                     //立即数
    int address;                //指令地址
    int dec_instruction_high; //十进制高16位指令
    char bin_instruction[34]; //二进制指令
    int id;                     //线程id
    struct Instrution *next;
} * LinkInstruction, Instruction;
```

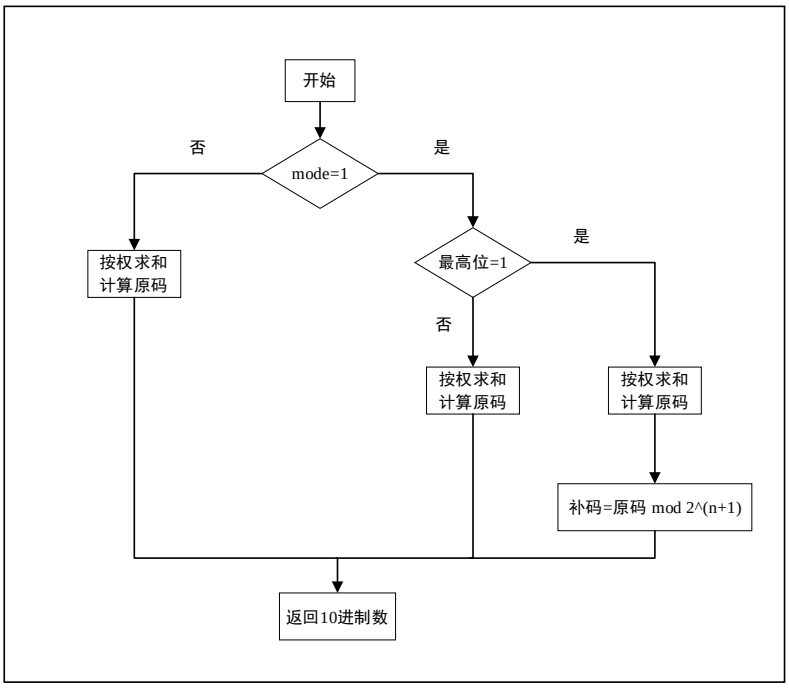
载入指令之后，main 函数调用两个线程函数，同时完成分析指令，执行指令。

在每个线程中，首先将指针指向头结点，当指针不为 NULL，则将指令从内存存入高速缓存器 cache 中，然后将指令的前 16 位存入指令寄存器 ir 中，程序计数器 pc 指向下一指令，然后分析指令并执行指令。执行完成后，通过 pc 的值判断程序的下一条指令，继续上述操作。并且程序每执行完一条，打印寄存器状态，直到读取到停机指令。

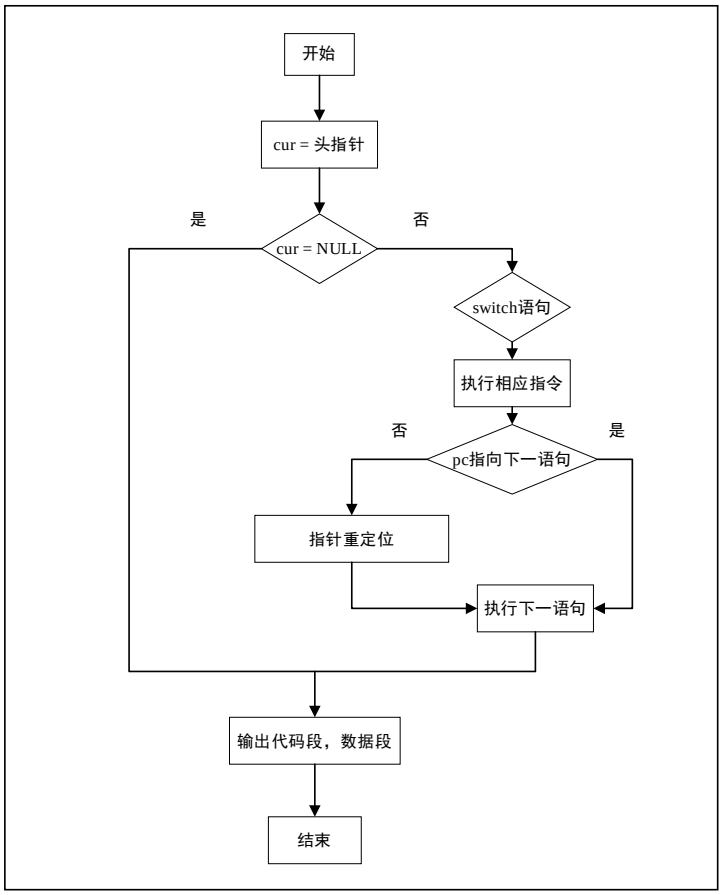
最后打印此时内存中的代码段和数据段，程序到此完全结束。

核心代码

1. 2 进制转 10 进制算法



2. 指令执行控制算法



3. 程序设计细节

- 内存锁问题

由于请求互斥对象语句可以锁住任何后续的操作内存，直到释放互斥对象，所以使用互斥对象实现内存锁功能。WaitForSingleObject 与 ReleaseMutex 的配合使用，可以实现互斥对象的访问，从而使每一线程在对象变为有信号状态后才能访存。

- 寄存器选择问题

CPU 寄存器设计采用数组，下标 0 为线程 1 的寄存器，下标 1 为线程 2 的寄存器。在指令结点中存入线程 id，通过线程 id 控制访问不同线程的寄存器。使用数组结构能很容易地兼容更多核的 CPU 设计。

- 高速缓存器设计

为了模拟冯诺依曼体系计算机的工作情况，在 CPU 设计中加入 cache 设计。在单核版中，cache 被设置为全局变量，但是在多核版中，为了保持兼容性，将 cache 设置为局部变量，在个指令间传递。

教师评语：