计算机组成原理实验报告
13349051 劳嘉辉 计科一班

实验内容

使用 C++或者 C 语言生成一个程序。这个程序相当于 FSM 的模拟器。模仿机器在执行 LC2K 指令时，datapath 中数据流动的情况。这是一个更详细、抽象、低层的一个层次。

实验目的

1、 更好地了解 LC2K 汇编语言
2、 更好地了解机器码是如何执行的
3、 更好地了解到机器码执行的过程中内存 memory 和寄存器 reg 内容的变化。
4、 更好地了解到机器码执行的过程中数据是如果通过 system bus 传递到不同的内存和寄存器中的。

代码显示

1、定义内存寄存器结构体

```
typedef struct stateStruct {
    int pc;
    int mem[NUMMEMORY];
    int reg[NUMREGS];
    int memoryAddress;
    int memoryData;
    int instrReg;
    int aluOperand;
    int aluResult;
    int numMemory;
} stateType;
```

这个 stateStruct 模拟一个机器内部的容器。Pc 指的就是 program control 程序控制，用于判断目前执行到哪一个步骤。而 mem[]则是模拟一个连续的内存空间，里面装的是 instruction 或者是 offset。Reg[]则是模拟一个寄存器空间，里面存放的是模拟寄存器的值。Men[]和 reg[]在一开始的时候都会被 memset()函数清空为 0。而 numMemory 则是用来该程序需要多少的空间。instrReg 则是存储指令的寄存器，aluOperand 则是 alu 的第一个操作数，aluResult 则是 alu 的结果。

2、打印内存寄存器结构体内部情况的函数

```
void
printState(stateType *statePtr, char *stateName)
{
    int i;
    static int cycle = 0;
    printf("\n@@@\nstate %s (cycle %d)\n", stateName, cycle++);
    printf("\tpc %d\n", statePtr->pc);
    printf("\tmemory:\n");
    for (i = 0; i<statePtr->numMemory; i++) {
        printf("\t\tmem[ %d ] %d\n", i, statePtr->mem[i]);
    }
    printf("\tregisters:\n");
    for (i = 0; i<NUMREGS; i++) {
        printf("\t\treg[ %d ] %d\n", i, statePtr->reg[i]);
    }
    printf("\tinternal registers:\n");
    printf("\t\tmemoryAddress %d\n", statePtr->memoryAddress);
    printf("\t\tmemoryData %d\n", statePtr->memoryData);
    printf("\t\tinstrReg %d\n", statePtr->instrReg);
    printf("\t\taluOperand %d\n", statePtr->aluOperand);
    printf("\t\taluResult %d\n", statePtr->aluResult);
}
```

该函数主要是利用 struct 中的元素来展示该结构体内部有什么。然后用printf()函数展示出来。

3、signed 数转换为 unsigned 数

```
Int convertNum(int num)
{
    /* convert a 16-bit number into a 32-bit Linux integer */
    if (num & (1 << 15)) {
        num -= (1 << 16);
    }
    return(num);
}
```

这是老师给的函数，这种做法我以前没有用过，确实很巧妙。这里是把二的十五次方与 num 比较，其实就是比较两个数字的首个字符，如果都是 1 的话，说明 num 其实一个负数，那么就把 num 减去二的十六次方从而得到相应的负数如果 num 的首个数字是 0，那么说明这个 num 是一个正数，那么就没有转换的必要了。

5、 内存操作函数

```
int
memoryAccess(stateType *statePtr, int readFlag)
{
    static int lastAddress = -1;
    static int lastReadFlag = 0;
    static int lastData = 0;
    static int delay = 0;

    if (statePtr->memoryAddress < 0 || statePtr->memoryAddress >= NUMMEMORY) {
        printf("memory address out of range\n");
        exit(1);
    }
    /*
    * If this is a new access, reset the delay clock.
    */
    if ((statePtr->memoryAddress != lastAddress) ||
        (readFlag != lastReadFlag) ||
        (readFlag == 0 && lastData != statePtr->memoryData)) {
        delay = statePtr->memoryAddress % 3;
        lastAddress = statePtr->memoryAddress;
        lastReadFlag = readFlag;
        lastData = statePtr->memoryData;
    }
    if (delay == 0) {
        /* memory is ready */
        if (readFlag) {
            statePtr->memoryData = statePtr->mem[statePtr->memoryAddress];
        }
        else {
            statePtr->mem[statePtr->memoryAddress] = statePtr->memoryData;
        }
        return(1);
    }
    else {
        /* memory is not ready */
        delay--;
        return(0);
    }
}
```

该函数模仿了在单周期过程中系统操作内存时发生的延迟现象。该延迟 delay 在本算法中最多为 2。如果 delay 为 0 的话，那么系统就可以成功读入或者写入内存。

6、 run()

该函数就是模拟单周期过程。

A、这里使用了 if-goto 的方法，从而可以不用使用 while,for 等指令来实现循环

```
infinteloop:
        body;
      if (true)
      goto infinteloop;
```

因为在执行 halt 指令的时候，程序会通过 exit(0)退出，所以并非是真的无限循环。

B、因为 memoryAccess()并非总能一次读取成功，因此同样使用 if-goto 的技巧代替 while,for

C、

Add 的数据流通过程：（R 型指令代表）

```
printState(&state, "add");
bus = state.reg[(state.instrReg - ((state.instrReg >> 22) << 22)) >> 19];
state.aluOperand = bus;
bus = state.reg[(state.instrReg - ((state.instrReg >> 19) << 19)) >> 16];
state.aluResult = state.aluOperand + bus;
bus = state.aluResult;
state.reg[(state.instrReg - ((state.instrReg >> 3) << 3))] = bus;
```

先通过 system bus 把 regA 的值读到 aluOperand 中，再通过 system bus 把 regB 的值读取，再通过 ALU 把 aluOperand+bus 的值读到 aluResult，再通过 system bus 把 aluResult 的值存到 destRegister 中。

Lw 数据流通过程：（I 型指令代表）

```
printState(&state, "lw");
bus = state.reg[(state.instrReg - ((state.instrReg >> 22) << 22)) >> 19];
state.aluOperand = bus;
bus = convertNum((state.instrReg - ((state.instrReg >> 16) << 16)) & 0xFFFF);
state.aluResult = state.aluOperand + bus;
bus = state.aluResult;
state.memoryAddress = bus;
load:if (!memoryAccess(&state, 1))
goto load;
bus = state.memoryData;//memorydata is stored in the system bus
state.reg[(state.instrReg - ((state.instrReg >> 19) << 19)) >> 16] = bus;
```

先通过 system bus 把 regA 的值读到 aluOperand 中，再通过 system bus 把 instrReg 中的 offset 读取出来，通过 alu 加法读到 aluResult，再通过 system bus 把 aluResult 的值读到 memoryAddress()中。

通过 if-goto 的方法读取 memoryAccess()直到读取成功。再通过 system bus 把

memoryData 读到 regB 中。

Jalr 指令的数据流通（J 型指令的代表）

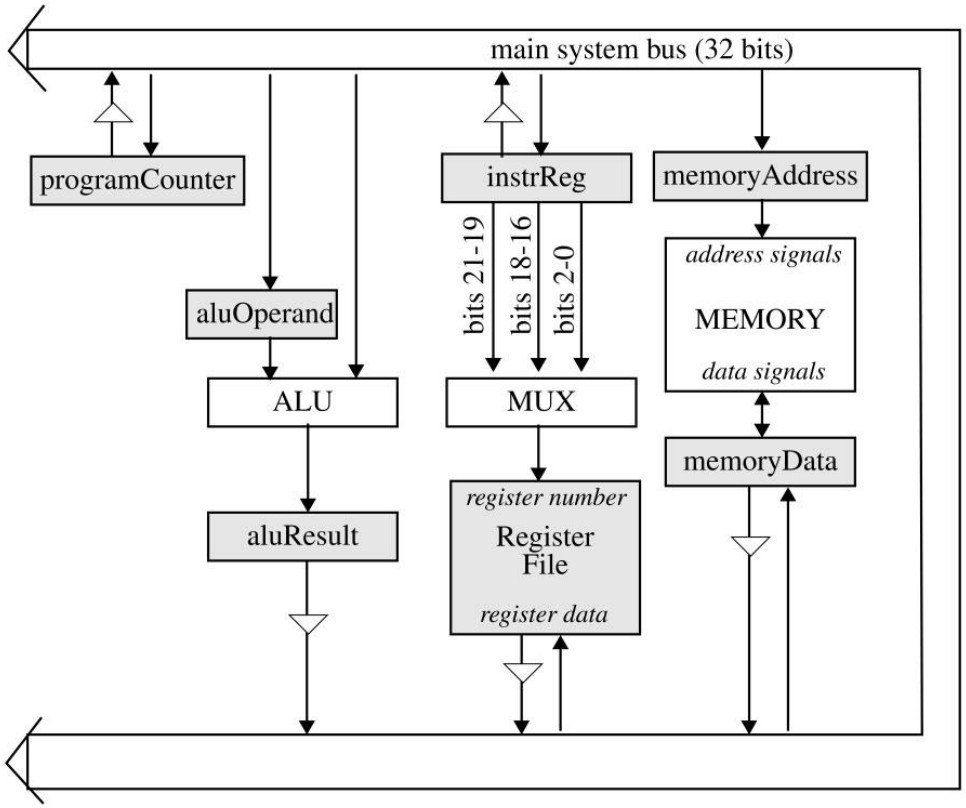
```
printState(&state, "jalr");
bus = state.pc;
state.reg[(state.instrReg - ((state.instrReg >> 19) << 19)) >> 16] = bus;
bus = state.reg[(state.instrReg - ((state.instrReg >> 22) << 22)) >> 19];
state.pc = bus;
```

先把 state.pc+1 的结果通过 system bus 传到 regB 中，再把 regA 中的值通过 system bus 传到 state.pc 中。

实验案例

实验案例较多，因此另外在文件夹里面显示。

实验图像



本实验严格按照本图中数据流通过程进行。

实验心得

本次实验与以前的实验不同。因为之前实现的 assembler 和 simulator 都是只要实现了相应的目的就可以了。但是这次的实验难点在于先要好好理解好在结构体系层下，finite state machine 运行的时候数据是如何从一个寄存器通过 system bus 到达另外一个寄存器或者内存的。而且本次实验的要求对打码也有相应的限制。这样使我可以从基础方面，一步一步地了解 finite state machine。这次的实验给了我不同的看法，我感觉收获良多。