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。

1. 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。

1. 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];

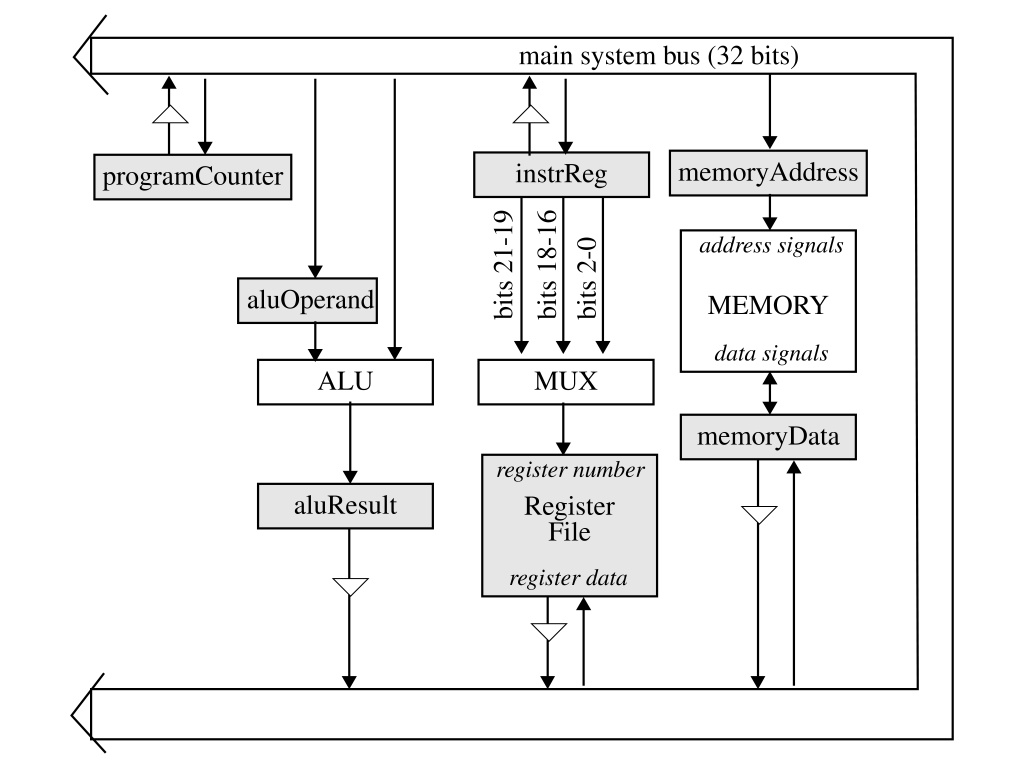
state.pc = bus;

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。