A Tomasulo Algorithm Simulator in C

Yihao Li
Northeast Normal University
Changchun, Jilin, China
liyh682@nenu.edu.cn

1 基本信息

1.1 项目描述

这是一个 Tomasulo 算法的 C 语言仿真器，源代码发布在 handsomedannyli/A-Tomasulo-Algorithm-Simulator-in-C: 2024 Spring NENU IST CS Computer Architecture Personal Topic (github.com)

1.2 工作量

代码行数：775 lines (593 loc)

1.3 使用方式

程序内预置了三个测试用例，在根目录下编译运行，输入要使用的测试案例，按任意键运行结果，查看测试结果。也可以自定义输入，修改代码中 MAX_INSTR_LENGTH 更改指令的最大长度，MAX_INSTRUCTIONS 更改最大指令数，REG_AMOUNT 更改浮点数寄存器数量，STATIONS_N 更改保留站数量，INSTRUCTION_AMOUNT 更改指令数量，并添加指令运行得到结果。

2 数据类型

2.1 指令类型

输入的汇编指令形式如下，本次 Tomasulo 算法仿真器应该支持识别并且读入以下 6 种汇编指令:

```
LD F6,34(R2)
SD F2,45(R3)
SUB F8,F2,F6
DIV F10,F0,F6
MUL F1,F3,F7
ADD F6,F8,F2
```

六种指令可以分成两大类，第一类指令是双寄存器指令 Fx,Rz 为寄存器，y 为立即数，指令形式为：

```
SD Fx,y(Rz)  LD Fx,y(Rz)
```

第二类指令是三寄存器指令，Fx,Fy,Fz 分别为三个寄存器，指令形式为：

ADD Fx,Fy,Fz SUB Fx,Fy,Fz DIV Fx,Fy,Fz MUL Fx,Fy,Fz

在这里，我使用两种结构体用来存储这两种指令，分别为 InstTwoReg 和 InstThreeReg

```
typedef struct
{
    int targetReg;
    int srcReg1;
    int offset;
} InstTwoReg;
//双寄存器指令，LD 和 SD

typedef struct
{
    int targetReg;
    int srcReg1;
    int srcReg2;
} InstThreeReg;
//三寄存器指令，ADD，SUB，MUL，DIV
```

用 int 数据类型存储目的寄存器号，源寄存器编号，和立即数。

一条指令不仅仅记录操作数和寄存器号，还需要记录指令的操作符以及指令的状态（流出的时间，开始的时间，结束的时间，写入的时间），因此完整的指令结构体定义如下：

```
typedef struct
{
    int operation;
    int type;
    InstThreeReg threeReg;
    InstTwoReg twoReg;
    //C 语言没有结构体，根据 type 的值决定使用 Instruction_ThreeReg 还是
Instruction_TwoReg
    int issuedT;
    int startedT;
    int finishedT;
    int writtenT;
    //流出，开始，结束，写入的时间点
} Instruction;
//指令结构体
```

用 int 类型变量 Operation 来计记录操作符，1-6 分别代表：LD,SD,ADD,SUB,MUL,DIV

用 int 类型变量 Type 来记录指令的类型，0 代表双寄存器指令，1 代表三操作数指令。

2.2 保留站类型

在 Tomasulo 算法中，保留站（Reservation Station）是用于处理指令操作的关键组件之一。保留站的主要作用是协调指令的执行，处理数据依赖，并管理指令的发射和执行。以下是保留站中的关键变量及其作用：

Operation (Op)用于存储当前指令的操作类型，如加法（ADD）、乘法（MUL）、加载（LD）等。

Target Register (Dest)用于指示操作结果将要写入的目标寄存器。

Source Operands (Src)用于存储操作所需的源操作数，可以是立即数或寄存器值。

变量：

Src1：第一个源操作数的值。

Src2：第二个源操作数的值。

Source Tags (Tags)用于指示源操作数的当前状态，如果操作数尚未准备好，它会存储数据所在的保留站标记。

变量：

depend1：指示第一个源操作数的保留站或功能单元标记。

depend2：指示第二个源操作数的保留站或功能单元标记。

Busy Flag 用于指示该保留站是否被占用。

Instruction（Inst）用于储存保留站内的指令，内含有 issued time 等信息

```
typedef struct{
    int busy;//unbusy:0;busy:1
    int type;//1:Load;2:Store;3:Adder;4:Multiplier
    Instruction instruction;
    int instIndex;
    int src1;
    int src2;
    int depend1;
    int depend2;
} Station;
```

用 int 数据类型存储 busy，0 为不忙 1 为忙，用 int 数据类型存储 type，1:Load;2:Store;3:Adder;4:Multiplier，station 里面有一个 Instruction 类型的指令，instIndex 是用于将保留站和指令在指令列表里关联起来，src 和 depend 分别记录了从源寄存器中获得的数值以及依赖保留站结果的保留站编号。

2.3 寄存器类型

在 Tomasulo 算法中，寄存器也扮演了非常重要的角色。为了实现指令的无序执行和解决数据依赖问题，寄存器状态需要进行特殊的管理。以下是寄存器中的关键变量及其作用：

Value 用于寄存器中存储的实际数据值。

Tag/Status 用于标记当前寄存器是否在等待数据，并且跟踪这个数据将由哪个保留站提供。

```
typedef struct
{
    int busyBy; // 用于指示当前占用该寄存器的保留站的索引
    int value;  // 寄存器中存储的数值
```

```
} Register;
```

使用 int 类型变量保存 busyBy，用于标记当前寄存器是否在等待数据，并且跟踪这个数据将由哪个保留站提供。使用 int 类型变量保存 Value，用于寄存器中存储的实际数据值。

3 数据获取与处理

3.1 指令的读入

指令输入为字符串，处理指令输入分成两步，先提取操作符，提取操作符转换成 int 类型，获取指令类型，是双寄存器类型还是三寄存器类型，对于指令的后半部分，根据指令的类型读取寄存器编号/立即数，将指令保存到 Instruction 类型的变量内。

提取操作符转换函数：

```
int getOp(char *op)
{
    if (strcmp(op, "LD") == 0)
        return 1;
    else if (strcmp(op, "SD") == 0)
        return 2;
    else if (strcmp(op, "ADD") == 0)
        return 3;
    else if (strcmp(op, "SUB") == 0)
        return 4;
    else if (strcmp(op, "MUL") == 0)
        return 5;
    else if (strcmp(op, "DIV") == 0)
        return 6;

    return -1; //获取失败
}
```

获取指令的类型函数：

```
int getType(int op)
{
    if (op == 1 || op == 2)
        return 0;
    if (op == 3 || op == 4 || op == 5 || op == 6)
        return 1;
}
```

使用以下函数提取指令后半段数据：

```
Instruction extractInstruction(char *instructionStr)
{
    Instruction rinstruct;
    char *regStr;
    char *opStr = strtok_r(instructionStr, " ", ®Str);

    //提取操作码和操作数部分

    int op = getOp(opStr);

    rinstruct.operation = op;
    rinstruct.type = getType(op);

    if(rinstruct.type == 0){
        sscanf(regStr, "F%d,%d(R%d)", &rinstruct.twoReg.targetReg,
&rinstruct.twoReg.offset, &rinstruct.twoReg.srcReg1);
    }
    //如果是双寄存器指令，twoReg 读入操作数

    if(rinstruct.type == 1){
        sscanf(regStr, "F%d,F%d,F%d", &rinstruct.threeReg.targetReg,
&rinstruct.threeReg.srcReg1, &rinstruct.threeReg.srcReg2);
    }
    //如果是三寄存器指令，threeReg 读入操作数

    rinstruct.issuedT = -1;
    rinstruct.startedT = -1;
    rinstruct.finishedT = -1;
    rinstruct.writtenT = -1;

    //时间初始化

    return rinstruct;
}
```

LD 和 SD 指令执行周期为 2，ADD 和 SUB 执行指令的周期为 3，MUL 执行指令所需周期为 10，DIV 指令执行所需周期为 40。

使用以下函数根据指令的操作数获取指令的执行时间：

```
int getOperationTime(int op){
    if (op == 1 || op == 2)
```

```
        return 2;
    if (op == 3 || op == 4 )
        return 3;
    if (op == 5)
        return 10;
    if (op == 6)
        return 40;
}
```

以下函数能够根据指令获得指令需要的保留站类型：LD（Load），SD（Store），ADD/SUB（Adder），MUL/DIV（Mutiplier）type：1:Load;2:Store;3:Adder;4:Multiplier

```
int getStationType(Instruction inst)
{
    if(inst.operation == 1){
        return 1;
    }
    if(inst.operation == 2){
        return 2;
    }
    if(inst.operation == 3 || inst.operation == 4){
        return 3;
    }
    if(inst.operation == 5 || inst.operation == 6){
        return 4;
    }
}
```

3.2 初始化

这一部分完成了参数初始化，在算法开始前完成保留站，寄存器，指令,时钟周期的初始化，并且读入指令。

寄存器 i 初始化为：registers[i].busyBy = -1;registers[i].value = -1;-1 代表空

保留站初始化为：{0, 1, 0, 0, -1, 1, 1, -1, -1}

Clock 促使化为 1，进入第一周期。

```
#define MAX_INSTR_LENGTH 25  // 指令的最大长度
#define MAX_INSTRUCTIONS 100 // 最大指令数
#define REG_AMOUNT 13        // 浮点数寄存器数量
#define STATIONS_N 9         // 保留站数量
#define INSTRUCTION_AMOUNT 6 // 指令数量

Void main(){
    Instruction instructionsList[INSTRUCTION_AMOUNT];
    for (int i = 0; i < 6; i++){
        instructionsList[i] = extractInstruction(inputInstructions[i]);
```

```
    }

    Register registers[REG_AMOUNT];
    for (int i = 0; i < REG_AMOUNT; i++){
        registers[i].busyBy = -1;
        registers[i].value = -1;
    }
    //初始化寄存器表

    Station reservationStations[STATIONS_N] = {
        {0, 1, 0, 0, -1, 1, 1, -1, -1},
        {0, 1, 0, 0, -1, 1, 1, -1, -1},
        {0, 2, 0, 0, -1, 1, 1, -1, -1},
        {0, 2, 0, 0, -1, 1, 1, -1, -1},
        {0, 3, 0, 0, -1, 1, 1, -1, -1},
        {0, 3, 0, 0, -1, 1, 1, -1, -1},
        {0, 3, 0, 0, -1, 1, 1, -1, -1},
        {0, 4, 0, 0, -1, 1, 1, -1, -1},
        {0, 4, 0, 0, -1, 1, 1, -1, -1}};

    //初始化保留站
    int clock = 1;
```

4 Tomasulo 算法实现

使用了一个主循环实现 Tomasulo 算法，循环内执行 Tomasulo 算法的核心三个步骤。

4.1 指令发射（Issue）

Tomasulo 算法是顺序发射的，即指令按照程序中的顺序一条接一条被发射到保留站。若还有指令没有发射，则尝试顺序发射指令，当指令从指令队列中取出时：

Step 1：检查保留站是否空闲：查找与指令操作类型匹配的空闲保留站。如果没有空闲的保留站，指令发射暂停。

```
if (currentPos < INSTRUCTION_AMOUNT) {

    // 尝试让该条指令流出
    int stationIndex = dispatchInstruction(reservationStations,
currentInstruction, clock, currentPos);

    if (stationIndex != -1) {

        // 更新指令的发射时间
```

```
        instructionsList[currentPos].issuedT = clock;
        currentPos += 1;

    }

}
```

Step 2：分配保留站：为指令分配一个空闲的保留站，并设置该保留站的 **Op** 字段为指令操作类型。

dispatchInstruction 函数会尝试将第 **currentPos** 位的指令分配给保留站，若分配成功，返回保留站的编号，失败返回**-1**.

```
int dispatchInstruction(Station reservationStation[STATIONS_N],
Instruction instruction, int clock, int instructionPosInLine)
{
    // 获取指令对应的保留站类型
    int stationType = getStationType(instruction);

    // 查找一个空闲的保留站
    int stationIndex = findNoBusyStation(reservationStation,
stationType);

    // 如果没有找到空闲的保留站，返回 -1
    if (stationIndex == -1)
        return -1;

    // 更新保留站的状态和指令信息
    reservationStation[stationIndex].busy = 1;
    reservationStation[stationIndex].instruction = instruction;
    reservationStation[stationIndex].instruction.issuedT = clock;
    reservationStation[stationIndex].instruction.startedT = -1;
    reservationStation[stationIndex].instruction.finishedT = -1;
    reservationStation[stationIndex].instruction.writtenT = -1;
    reservationStation[stationIndex].instIndex = instructionPosInLine;
    // 返回保留站的索引
    return stationIndex;
}
```

Step 3：周期结束时，更新保留站与寄存器状态表：检查指令所需的源操作数是否已准备好，如果源操作数已准备好（即寄存器的 **busyBy** 为 **-1**），将其 **Value** 直接存储在保留站的 **src1** 或 **src2** 中。如果源操作数尚未准备好（即寄存器的 **busyBy** 指向另一个保留站），将该标签存储在 **depend1** 或 **depend2** 中，并等待数据就绪。更新目标寄存器：将目标寄存器的 **busyBy** 设置为当前保留站的标记，以便后续指令知道该寄存器的值正在计

算中。

```
for (int i = 0; i < STATIONS_N; i++) {
    if (reservationStations[i].instruction.issuedT == clock) {

        if (reservationStations[i].busy == 0) {
            // 保留站不忙（busy 为 0），跳过该保留站，继续检查下一个
            continue;
        }

        // 初始化与当前指令相关的寄存器编号
        Instruction issued_inst = reservationStations[i].instruction;

        int instType = getType(issued_inst.operation);
        int targetReg = -1;
        int srcReg1 = -1;
        int srcReg2 = -1;

        int issued_inst_type = getType(issued_inst.operation);
        if (issued_inst_type == 0) {
            targetReg = issued_inst.twoReg.targetReg;
        }
        if (issued_inst_type == 1) {
            targetReg = issued_inst.threeReg.targetReg;
        }

        registers[targetReg].busyBy = i;

        if (issued_inst_type == 1) {
            srcReg1 =
reservationStations[i].instruction.threeReg.srcReg1;
            srcReg2 =
reservationStations[i].instruction.threeReg.srcReg2;

            if (isRegisterFree(registers[srcReg1], i) == 0 &&
registers[srcReg1].value == -1) {
                // src1 被占用
                reservationStations[i].src1 = 0;
                reservationStations[i].depend1 =
registers[srcReg1].busyBy;
            } else {
                reservationStations[i].depend1 = -1;
                reservationStations[i].src1 = registers[srcReg1].value;
            }
```

```
            if (isRegisterFree(registers[srcReg2], i) == 0 &&
registers[srcReg2].value == -1) {
                // src2 被占用
                reservationStations[i].src2 = 0;
                reservationStations[i].depend2 =
registers[srcReg2].busyBy;
            } else {
                reservationStations[i].depend2 = -1;
                reservationStations[i].src2 = registers[srcReg2].value;
            }
        }
    }
}
```

4.2 操作执行（Execute）

当保留站中所有源操作数都准备好时：

Step 1：检查操作数状态：保留站中 **dependl** 和 **depend2** 都为空（**-l**）时，说明所有操作数都准备好，可以执行。

Step 2：执行操作：根据操作类型执行计算，执行可能是多周期的，在执行过程中不改变处理器状态。当最后一个周期执行完，将 **writtenT** 赋值为 **clock+l**，表示在下一个周期写回。

```
for (int i = 0; i < STATIONS_N; i++) {

    printf("station:%s | busy:%d\n", stationName[i],
reservationStations[i].busy);

    // 如果保留站不忙（busy 为 0），跳过该保留站，继续检查下一个。
    if (reservationStations[i].busy == 0) {
        // printf("保留站不忙（busy 为 0），跳过该保留站，继续检查下一个
\n");
        continue;
    }

    // 如果这个周期刚流出，不执行
    if (reservationStations[i].instruction.issuedT == clock) {
        // printf("这个周期刚流出，不执行\n");
        continue;
    }


    // 获取指令类型
```

```
    int instType =
getType(reservationStations[i].instruction.operation);


    // 处理保留站中已准备好执行的指令，检查指令的状态和相关寄存器的使用情况，
决定是否可以开始执行指令
    if (reservationStations[i].instruction.startedT == -1) {

        if (isRegisterFree(registers[targetReg], i) == 0) {
            // printf("target 被占用\n");
            continue;
        }

        if (instType == 1) {
            if (reservationStations[i].depend1 != -1 &&
reservationStations[i].src1 == 0) {
                // printf("src1 被占用\n");
                continue;
            }
            if (reservationStations[i].depend2 != -1 &&
reservationStations[i].src2 == 0) {
                // printf("src2 被占用\n");
                continue;
            }
        }

        // 开始执行指令
        reservationStations[i].instruction.startedT = clock;
        instructionsList[reservationStations[i].instIndex].startedT =
clock;
        // printf("开始执行指令\n");
        continue;
    }

    // 检查指令是否已经完成
    if ((reservationStations[i].instruction.startedT +
getOperationTime(reservationStations[i].instruction.operation) - 1) <=
clock
        && reservationStations[i].instruction.writtenT == -1) {
        // 开始时间 + 执行时间 <= 当前时钟周期，说明指令已经完成
        reservationStations[i].instruction.finishedT = clock;
        instructionsList[reservationStations[i].instIndex].finishedT =
clock;
        reservationStations[i].instruction.writtenT = clock + 1;
```

```
        // 指令完成在下个周期写回
        continue;
    }
}
```

4.3 写回（Write-Back）

当指令完成执行后：

Step 1：通过 CDB 总线广播：检查目标寄存器的标签是否仍指向当前保留站。如果是，将结果写回目标寄存器，并将其 **busyBy** 设置为 **-1**。检查保留站内部是否还有需要该结果的数据依赖，如果有，将结果赋给 **src** 并将 **depend** 置为空（**-1**）.

Step 2：清空保留站：清空保留站，将 **Busy** 标志置为 **0**，将其他数据清空为下一条指令准备。

```
for (int i = 0; i < STATIONS_N; i++) {

    if (reservationStations[i].instruction.writtenT == clock) {
        int targetReg = -1;

        // 获取指令类型
        int instType =
getType(reservationStations[i].instruction.operation);

        if (instType == 0) {
            targetReg =
reservationStations[i].instruction.twoReg.targetReg;
        }
        if (instType == 1) {
            targetReg =
reservationStations[i].instruction.threeReg.targetReg;
        }

        // 写回，释放 targetReg，释放保留站
        if (registers[targetReg].busyBy == i) {
            registers[targetReg].busyBy = -1;
            registers[targetReg].value =
reservationStations[i].instIndex + 1;
        }

        // 广播更新保留站的源寄存器
        for (int j = 0; j < STATIONS_N; j++) {
            if (reservationStations[j].src1 == 0 &&
reservationStations[j].depend1 == i) {
                reservationStations[j].src1 =
reservationStations[i].instIndex + 1;
                reservationStations[j].depend1 = -1;
```

```
            }
            if (reservationStations[j].src2 == 0 &&
reservationStations[j].depend2 == i) {
                reservationStations[j].src2 =
reservationStations[i].instIndex + 1;
                reservationStations[j].depend2 = -1;
            }
        }

        // 释放保留站
        reservationStations[i].busy = 0;
        reservationStations[i].src1 = 0;
        reservationStations[i].src2 = 0;
        reservationStations[i].depend1 = -1;
        reservationStations[i].depend2 = -1;
        instructionsList[reservationStations[i].instIndex].writtenT =
clock;

        continue;
    }

    // 将 targetReg 设置为空闲，释放保留站
}
```

5 输出以及测试

5.1 输出

在每个周期结束之前，使用控制台分别输出指令状态表，保留站，寄存器状态表。这部分内容过于冗长，具体实现流程就是严格控制输出字符串的长度，达到输出表格的效果。

5.2 测试

程序内置了三个测试用例，分别涵盖了 **RAW,WAW,WAR** 三种数据依赖情况，使用课程群里发布的 **Tomasulo** 算法仿真器对测试用例生成标准答案，并人工将答案转换到和本次输出相同的格式上，使用 **C** 语言对两个文件的内容进行逐行比较。输出比较结果

```
#include <stdio.h>

int compareFiles(const char *file1, const char *file2) {
    FILE *fp1 = fopen(file1, "rb");
    FILE *fp2 = fopen(file2, "rb");

    if (fp1 == NULL || fp2 == NULL) {
        perror("Error opening file");
```

```
        if (fp1) fclose(fp1);
        if (fp2) fclose(fp2);
        return -1;
    }

    int ch1, ch2;
    int position = 0;

    do {
        ch1 = fgetc(fp1);
        ch2 = fgetc(fp2);
        position++;

        if (ch1 != ch2) {
            printf("Files differ at byte %d\n", position);
            fclose(fp1);
            fclose(fp2);
            return 1;
        }
    } while (ch1 != EOF && ch2 != EOF);

    if (feof(fp1) && feof(fp2)) {
        printf("Files are identical.\n");
        fclose(fp1);
        fclose(fp2);
        return 0;
    } else {
        if (!feof(fp1)) {
            printf("File 1 has extra content after byte %d.\n",
position);
        } else if (!feof(fp2)) {
            printf("File 2 has extra content after byte %d.\n",
position);
        }
        fclose(fp1);
        fclose(fp2);
        return 1;
    }
}

int main() {
    const char *file1 = "out.txt";
    const char *file2 = "ans.txt";
```

```
    int result = compareFiles(file1, file2);
    if (result == 0) {
        printf("The output is correct.\n");
    } else if (result == 1) {
        printf("The output is incorrect.\n");
    }

    return result;
}
```

结果正确