Tomasulo 实验报告

计04 何秉翔 2020010944

1. commit hash

最终的 `git commit hash` 为：`640777c2f32718345ad53c24fbaa030c1373dd77`，在 `distribute` 分支上。

2. 必做功能

我们所实现的功能如下：

2.1 保留站插入指令

即 `ReservationStation<size>::insertInstruction` 方法，我们遍历所有的发射槽，若有空闲的槽位，我们在该槽位插入指令，更新 `inst`、`robIdx` 字段，并设为 `busy`，接着我们分别对指令的 `rs1` 和 `rs2` 来设置 `readPort` 读端口，具体而言：

1. 先看寄存器是否 `busy`，若不 `busy`，则说明值已经更新到寄存器，可直接调用寄存器的 `read` 接口读出值，并设置读端口不需要等待。
2. 若 `busy`，则说明值未更新到寄存器，首先我们需要获得 `busy` 的那个寄存器对应的 `robIdx`，并以此来调用 `checkReady` 方法来看是否 `rob` 已经获得值，若未 `ready`，说明值未就绪，我们设置读端口等待，并更新所等待的 `robIdx`；否则值已经就绪，调用 `rob` 的 `read` 方法读出值，并设置读端口不需要等待。

2.2 保留站唤醒

即 `ReservationStation<size>::wakeup` 方法，我们遍历所有的发射槽，对每个槽位，分别对两个读端口判断，若读端口正在等待，并且所等待的 `robIdx` 与 `ROBStatusWritePort` 给的 `robIdx` 一致，则我们更新读端口的值，并设置不需等待。

2.3 保留站判断是否能发射指令

即 `ReservationStation<size>::canIssue` 方法，我们遍历所有的发射槽，尝试找到第一个不 `busy` 并且两个端口都值已就绪的发射槽，如果找得到返回 `true`，否则找不到返回 `false`。

2.4 保留站发射指令

即 `ReservationStation<size>::issue` 方法，我们遍历所有的发射槽，找到第一个不 `busy` 并且两个端口都值已就绪的发射槽，将该槽的信息返回，并将原来该槽位的 `busy` 置空。

2.5 后端分配指令

即 `Backend::dispatchInstruction` 方法，原有代码已经判断 `rob` 是否已满，因此我们只需获得对应指令的类型，分别对六种类型的指令 `ALU`、`BRU`、`DIV`、`LSU`、`MUL`、`NONE` 进行后续操作，具体而言，对于非 `NONE` 指令，我们分别：

1. 通过 `hasEmptySlot` 接口检查对应的保留站是否已满，满了直接返回 `false`。
2. 通过 `push` 方法将指令插入 `rob`，设置 `ready` 为 `false`，并获得 `robIdx`。
3. 接着将指令插入保留站。
4. 最后更新 `rd` 寄存器占用情况，设置为 `busy`。

对于 `NONE` 类型指令，我们只插入 `rob`，不插入保留站，直接 `ready`。

2.6 后端提交指令

即 `Backend::commitInstruction` 方法，我们从 `rob` 中弹出需要提交的指令，获得 `robIdx`，接着我们同样根据指令的类型来进行不同的操作，判断写入寄存器 `rd` 结果，对于 `Store` 类型指令，我们还写入 `data`，注意需要进行偏移 `(slot.storeAddress - 0x80400000) >> 2` 写入。

最后对于 `BRU` 类型指令，我们针对 `mispredict` 的情况调用前端的 `jump` 方法进行跳转，具体而言，`mispredict` 分以下两种情况（加入 `BTB`）：

1. 预测不分支，但是通过 `actualTaken` 判断实际上分支了，跳转到分支地址 `entry.state.jumpTarget`
2. 预测分支，但是通过 `actualTaken` 判断实际上不分支，跳转到 `pc + 4`

最后调用 `flush` 清空流水线。

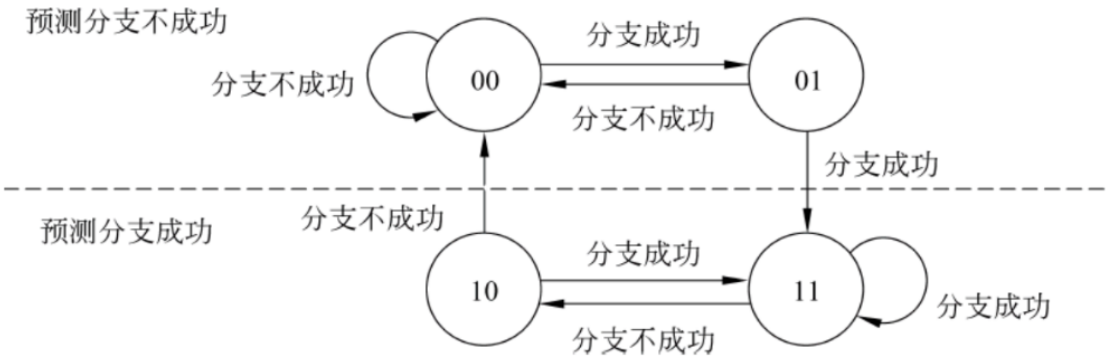
3. 选做功能

3.1 `BTBEntry` 设计

我们首先设计 `BTBEntry` 如下：

```
struct BTBEntry {
    unsigned pc;     // 分支指令地址
    unsigned bht;    // 分支历史表，取值 0，1，2，3，分别对应 00、01、10、11，与实验文档语义一致
    unsigned target; // 分支目标地址
    bool valid;      // 该 entry 是否有值
};
```

接下来的部分我们按照实验文档图设计：



3.2 初始化 `BTB`

我们设置 `BTB` 为直接相连，共 `1024` 个槽位，放在 `FrontendWithPredict` 下，我们直接遍历 `BTB`，给每一个槽位置为 `invalid` 即可。

```
FrontendWithPredict::FrontendWithPredict(const std::vector<unsigned> &inst)
    : Frontend(inst) {
    // Optional TODO: initialize your prediction structures here.
    for (int i = 0; i < 1024; ++i) {
        btb_table[i].valid = false;
    }
}
```

3.3 获取指令的分支预测结果

由于 `pc` 按照 4 进行偏移，我们在映射到 `BTB` 时采用 `(pc >> 2) % 1024` 来确定对应 `BTB` 槽位，对于 `FrontendWithPredict::bpuFrontendUpdate` 方法，我们返回 `BranchPredictBundle` 对象，通过找到对应的 `BTB` 槽位，若分支指令地址一致并且该位置是 `valid` 的，然后我们根据 `BHT` 来预测分支是否成功：

```
BranchPredictBundle FrontendWithPredict::bpuFrontendUpdate(unsigned int pc) {
    // Optional TODO: branch predictions
    BranchPredictBundle result;
    unsigned i = (pc >> 2) % 1024;
    if (btb_table[i].valid && btb_table[i].pc == pc) {
        if (btb_table[i].bht == 0 || btb_table[i].bht == 1) {
            // 预测分支不成功
            result.predictJump = false;
        } else { // 2 or 3
            result.predictJump = true;
            result.predictTarget = btb_table[i].target;
        }
    } else {
        result.predictJump = false;
    }
    return result;
}
```

3.4 计算 NextPC

基本思路与上面方法一致，找到对应槽位，根据 `BHT`，如果预测分支成功，则返回 `BTB` 里的分支目标地址，否则返回 `pc + 4`。

```
unsigned FrontendWithPredict::calculateNextPC(unsigned pc) const {
    // Optional TODO: branch predictions
    unsigned i = (pc >> 2) % 1024;
    unsigned next_pc = 0;
    if (btb_table[i].valid && btb_table[i].pc == pc) {
        if (btb_table[i].bht == 0 || btb_table[i].bht == 1) {
            // 预测分支不成功
            next_pc = pc + 4;
        } else { // 2 or 3
            next_pc = btb_table[i].target;
        }
    } else {
        next_pc = pc + 4;
    }
    return next_pc;
```

3.5 更新分支预测器状态

我们首先找到对应的槽位，如果对应槽位 `valid` 并且分支指令地址一致，则根据实际分支成功与否，以及 `BHT` 来更新 `BHT` 状态；否则在该槽位插入一条新的分支信息，更新 `BTBEntry` 相应字段：

```
void FrontendWithPredict::bpuBackendUpdate(const BpuUpdateData &x) {
    // Optional TODO: branch predictions
    unsigned i = (x.pc >> 2) % 1024;
    if (btb_table[i].valid && btb_table[i].pc == x.pc) {
        btb_table[i].target = x.jumpTarget;
        if (x.branchTaken) { // 分支成功
            if (btb_table[i].bht == 0) { // 00 -> 01
                btb_table[i].bht = 1;
            } else { // 01 10 11 -> 11
```

```
                btb_table[i].bht = 3;
            }
        } else { // 分支失败
            if (btb_table[i].bht == 3) { // 11 -> 10
                btb_table[i].bht = 2;
            } else { // 00 01 10 -> 00
                btb_table[i].bht = 0;
            }
        }
    } else {
        btb_table[i].pc = x.pc;
        btb_table[i].valid = true;
        btb_table[i].target = x.jumpTarget;
        if (x.branchTaken) {
            btb_table[i].bht = 1;
        } else {
            btb_table[i].bht = 0;
        }
    }
}
```

3.6 更新后端指令提交

我们在 `Backend::commitInstruction` 中对于分支指令调用前端的 `bpuBackendUpdate` 方法，在分支指令提交时根据实际分支情况来更新 `BTB`：

```
BpuUpdateData x;
x.branchTaken = entry.state.actualTaken; // 实际上分支成功
x.jumpTarget = entry.state.jumpTarget;
x.pc = entry.inst.pc;
frontend.bpuBackendUpdate(x);
```