各个 Stage 的工作

Pipeline 有多种可能的实现，表 1 说明了本模拟器机制下各个 Stage 的工作。

表 1 各个 Stage 的工作

Stage	工作
IF	获取指令并解析。更新 pc 寄存器，对于 Branch 指令需做出分支预测。
ID	读取 src 寄存器或 Forwarding 数据，获取正确的右值或发起 Stall。
EX	执行计算或分支判断，在分支预测失败时修改 pc 寄存器，并发起 Bubble。
MEM	对于 Load 和 Store 指令，执行内存读写。
WB	将指令结果写入 dst 寄存器。

指令集类的设计

对于每一条指令，需要五个执行函数 `pc_modify`、`inst_decode`、`execute`、`mem_access`、`write_back`。为了用统一的代码调度不同指令的执行函数，利用 C++多态特性，将指令封装为类，继承自统一的父类 `Inst`，并将五个执行函数设计为 `Inst` 类的虚函数。相同类别下的不同指令的一些执行函数是相同的，利用继承机制实现去重。

为了实现 Forwarding 机制，在 `Inst` 类中添加了 `forward` 虚函数，用于提供 Forwarding 数据（或发起 Stall）。并对有读寄存器需求的指令类添加 `get_fwd` 函数，以请求 Forwarding 数据。

Parser 的设计

获取二进制代码后，利用 `Inst::parse` 函数解析 `opcode`，判断指令大类，然后将二进制代码传入 `RTypeInst::parse`、`ITypeInst::parse` 等函数进一步解析。

简化 Pipeline Register

在 Pipeline 运行的过程中，一些数据被存入 Pipeline Register，以使得 IF 以外的四个 Stage 可以获得运行所需的数据，这些数据主要包括

1. 除 IF 以外的 Stage 正在运行的指令
2. （某个 Stage 正在运行 Branch 指令或 JALR）当前指令的 pc、预测的下一指令 pc
3. （某个 Stage 正在运行需要寄存器读取的指令）ID 阶段读取到的右值
4. （某个 Stage 正在运行需要计算的指令）EX 阶段的计算结果
5. （某个 Stage 正在运行 Load 指令）MEM 阶段内存读取结果

在软件模拟 Pipeline 时，为了简化代码，提高运行速度，将这些数据视作指令类的成员变量。创建 `Inst * inst[5]`数组，每个 Stage 通过 `Inst *`指向的对象了解正在执行的指令类型，并获得执行指令所需的数据。

Forwarding

利用两个函数 `forward` 和 `get_fwd` 实现了 Forwarding 机制。其中，`forward` 函数提供 Forwarding 数据，`get_fwd` 函数请求 Forwarding 数据。

```
bool Inst::forward(Stage stage, unsigned src, unsigned & rval)
```

当前指令所处的 Stage 为 `stage`，请求 Forwarding 的指令需要从编号为 src 的寄存器中读取右值，当前指令提供的 Forwarding 数据存入 `rval`，返回的 `bool` 类型值表示是否发生了 Forwarding。在 Load 指令的 `forward` 函数中，若 `stage==EX`，则发起 Stall。

```
void SrcInst::get_fwd(unsigned src, unsigned & rval)
```

当前指令需要从编号为 src 的寄存器中读取右值，该右值应存入 rval。该函数会依次向处于 EX、MEM、WB 的指令请求 Forwarding，一旦某一请求成功，函数结束。

分支预测

`Predictor` 类实现了二级自适应分支预测器，分支预测准确率实验结果如表 2 所示。

表 2 分支预测准确率

测试数据	准确率
array_test1	54.55%
array_test2	50.00%
basicopt1	98.89%
bulgarian	94.50%
expr	76.58%
gcd	61.67%
hanoi	86.48%
lvalue2	66.67%
magic	82.20%
manyarguments	80.00%
multiarray	68.52%
naive	未执行分支
pi	84.51%
qsort	91.74%
queens	81.01%
statement_test	66.83%
superloop	95.19%
tak	75.70%