lab4 实验报告

雷雨轩 PB18111791 计算机学院

实验目的

- 实现BTB（Branch Target Buffer）和BHT（Branch History Table）两种动态分支预测器
- 体会动态分支预测对流水线性能的影响

实验环境

- Vscode 2021
- Vivado 2019.1

实验内容

- 阶段一：在Lab3阶段二的RV32I Core基础上，实现BTB
- 阶段二：实现BHT
- 阶段二**需要在阶段一的基础上实现**，不能仅实现阶段二
- 阶段三：提供了btb.s、bht.s、QuickSort.s、MatMul.s四个测试样例，分别执行这四个测试样例，并在报告中分析以下内容：
    - 分析分支收益和分支代价
    - 统计未使用分支预测和使用分支预测的总周期数及差值
    - 统计分支指令数目、动态分支预测正确次数和错误次数
    - 对比不同策略并分析以上几点的关系

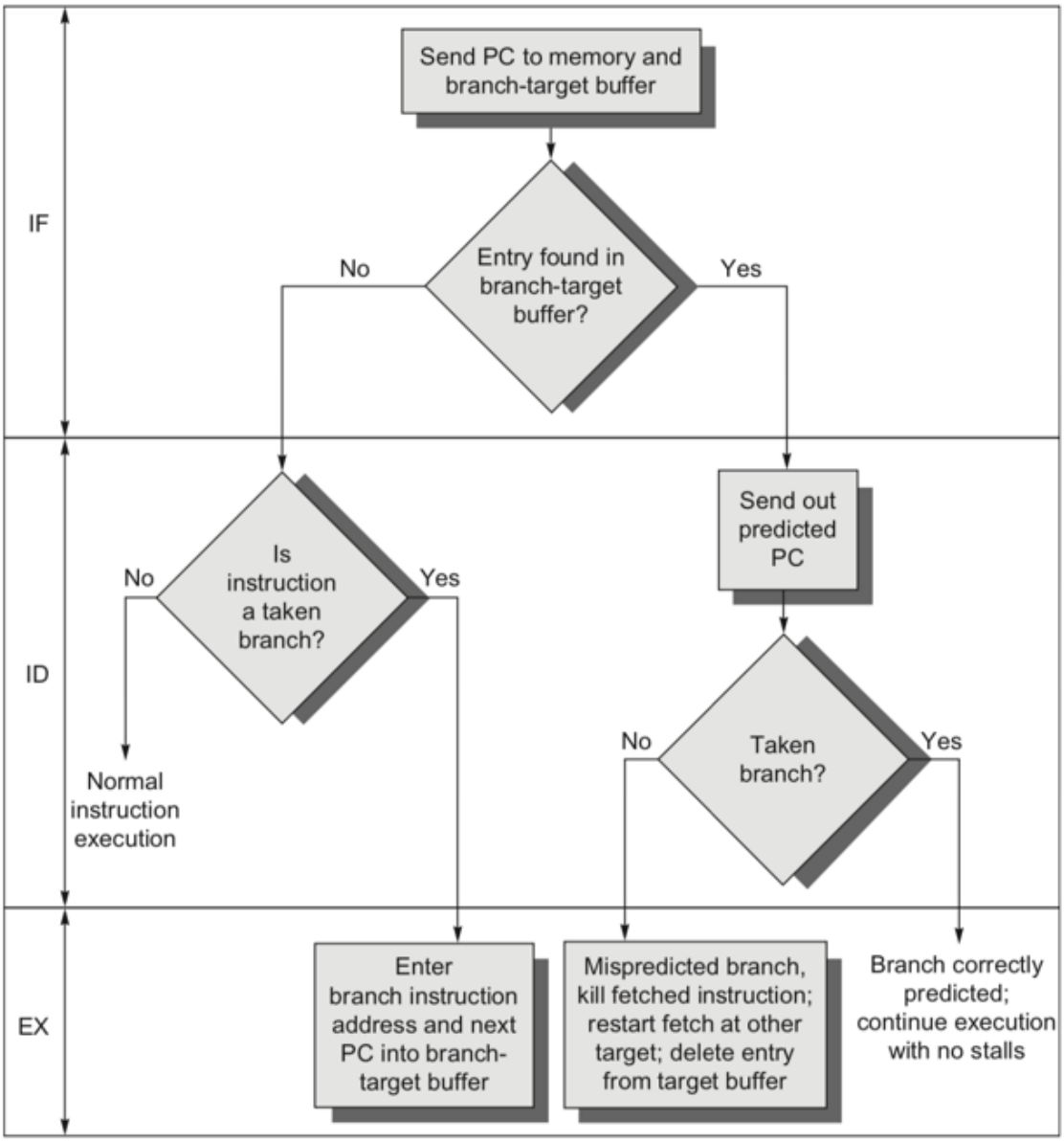
实验过程

阶段1

实现思路

- 在添加了BTB之后，对于IF阶段产生的PC
    - 在BTB Buffer里检查是否有对应项，如果有的话，选择predicted PC作为下一PC。
    - 如果当前PC不在BTB表里，则预测不跳转。
    - 若在EX段发现是一条需要跳转的Branch指令，则在EX阶段更新BTB表（当BTB表里没有该项时，则增加该项），否则，有该项，则保持不变
    - 若在EX段发现是一条不跳转的Branch指令，如果PC在BTB表中，也需要更新BTB表（将该指令从buffer里去掉，即清0），并flush错误装载的指令

- 状态机如图：



BTB.sv

- 首先是考虑需要的输入输出信号

```
input  clk, //时钟信号
input  rst,                          //复位信号
input [31:0] PCRead,                 //输入PC，即当前读到的PC，要据此来判断下一条指令是什么
input btb_w,                         //写请求
input [31:0] PCWrite,                //输入指令，即有更新BTB要求的指令PC值
input [31:0] Predict_PCW,            //对应需要写入的PC值
input StateW,           //对应的预测状态位

output reg ReadPredict,              //输出状态，为 1 表示PCRead是Branch指令，对应Predict_PCR 是预测的PC
output reg [31:0] Predict_PCR   //从 buffer 中读出的预测 PC
```

一个是根据当前IF段的PC值去BTB表里读取对应项（或者说miss）；另一个则是根据当前EX段的PC值来更新BTB

- 再参考直接相连cache里的实现，考虑对PC值做切分，高位用于与buffer内容进行比较，即tag；低位用于buffer内寻址，并且由于直接相连映射，每个PC值在buffer里有唯一地址。当然最低的2个bit空出来。
- 对读取buffer的情况，需要先与BTB表内项内容命中，再看是否预测跳转

```
always @(*)
begin
    //判断输入的 PC 是否在 buffer 中命中
    if(PCTag[ReadBufferAddr]==ReadTagAddr && StateBit[ReadBufferAddr])//如果tag与输入地址中的tag部分相等且buffer的该项有效，则命中
        ReadPredict = 1'b1;
    else
        ReadPredict = 1'b0;
    Predict_PCR=Predict_PC[ReadBufferAddr];
end
```

- 对EX段更新buffer的情况，即需要对Branch指令做判断，若其预测和实际执行情况不同，那么需要更新BTB，更新方式为直接覆盖对应的BTB表项，并且预测跳转与否改为与实际情况一致

```
if(btb_w)
begin
    Predict_PC[WriteBufferAddr] <= Predict_PCW;
    PCTag[WriteBufferAddr] <= WriteTagAddr;
    StateBit[WriteBufferAddr] <= StateW;
end
```

NPC_Generator.v

- 需要特别注意，在添加了分支预测的功能后
  - 首先是IF段如果BTB预测跳转，那么需要使用预测PC值
  - 如果EX段发现是Branch指令，并且之前的预测与实际的结果不一致，那么需要做恢复
  - 并且需要根据段顺序反向进行if语句判断

```
always@(*)
    begin
        //if语句判断的顺序体现了优先级，EX阶段的优先级高于ID段,具体原因可参考lab1报告的分析
        if (JalrE)
            PC_In <= JalrTarget;
        else if(BranchE && ~PredictedE) // 预测不跳转，实际跳转
            PC_In <= BranchTarget;
        else if(~BranchE && PredictedE) // 预测跳转，实际不跳转
            PC_In <= PCE + 4;
        else if(JalD)
            PC_In <= JalTarget;
        else if(PredictedF) // 预测即跳转
            PC_In <= PredictedTargetF;
        else
            PC_In <= PCF + 4;
    end
```

IDSegReg.v

- 只需要正常的把Predicted信号传递下去即可

ExSegReg.v

- 只需要正常的把Predicted信号传递下去即可

HazardUnit.v

- 需对Branch指令特别判断，如果 EX 段发现预测情况与实际情况不一致，需要flush
  - 预测跳转但实际不跳
  - 预测不跳转但实际要跳转
  - 当BTB预测与实际执行情况一致时，是完全正常的执行，所以不需要flush

```
else if((BranchE ^ PredictedE) | JalrE)//当在EX段检测到跳转信号时，根据lab1里分析，应该把ID和EX段寄存器清空;但由于有预测分支情况，需作出改变

 {StallF,FlushF,StallD,FlushD,StallE,FlushE,StallM,FlushM,StallW,FlushW} <= 10'b0001010000;
```

RV32Core.v

- 首先即是对上述提到修改的模块的接口作一一对接
- 其次需要增加一个计数模块，因为阶段3里统计信息的需要

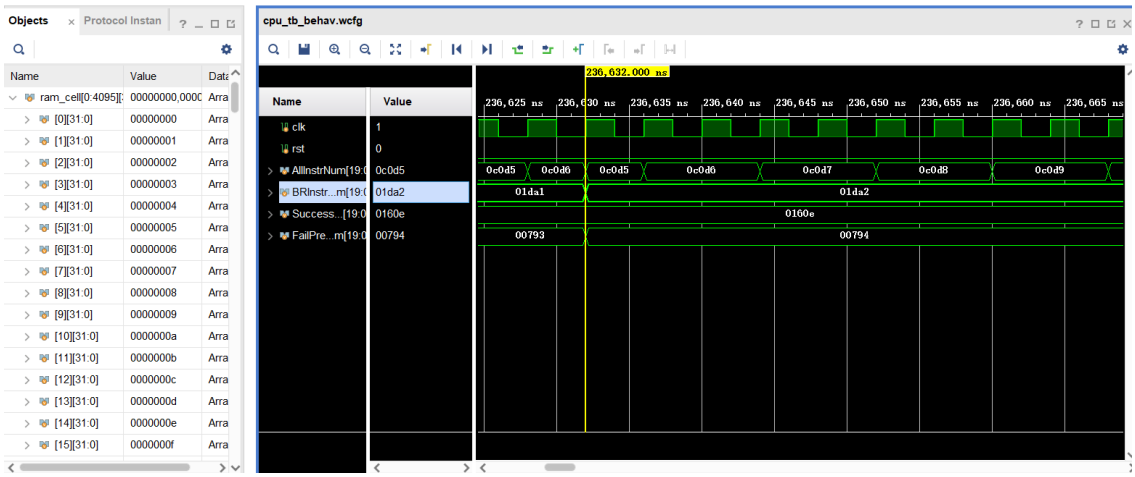
```
reg [19:0] AllInstrNum;//运行的总指令数
reg [19:0] BRInstrNum;//分支指令数
reg [19:0] SuccessPredictNum;//成功预测的分支数
reg [19:0] FailPredictNum;//预测失败的分支数

always @(posedge CPU_CLK or posedge CPU_RST)
    begin
        if(CPU_RST)
        begin
            AllInstrNum <= 0;
            BRInstrNum <= 0;
            SuccessPredictNum <= 0;
            FailPredictNum <= 0;
        end
        else
        begin
            if(FlushD && FlushE)
                AllInstrNum <= AllInstrNum - 1;
            else if(FlushD || FlushE)
                AllInstrNum <= AllInstrNum;
            else
                AllInstrNum <= AllInstrNum + 1;
            if(BranchTypeE != 3'b000)
            begin
                BRInstrNum <= BRInstrNum + 1;
                if(PredictedE ^ BranchE)
                    FailPredictNum <= FailPredictNum + 1;
                else
                    SuccessPredictNum <= SuccessPredictNum + 1;
            end
        end
```

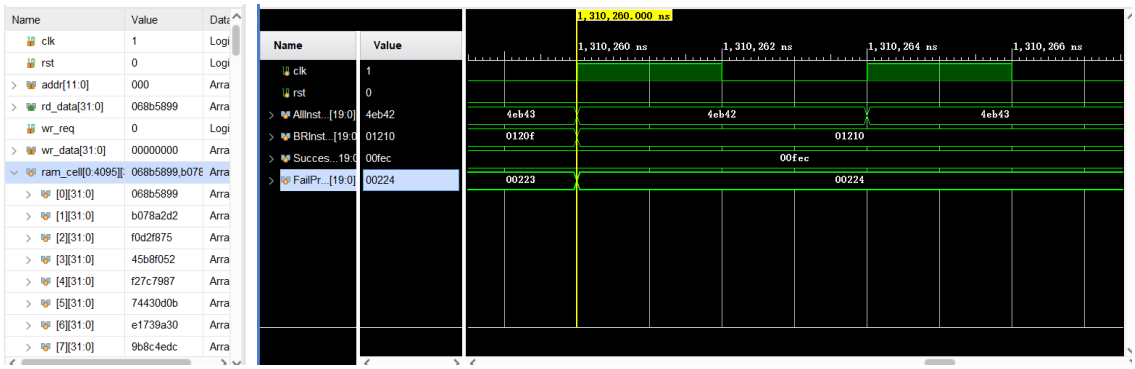
```
    end
```

运行结果

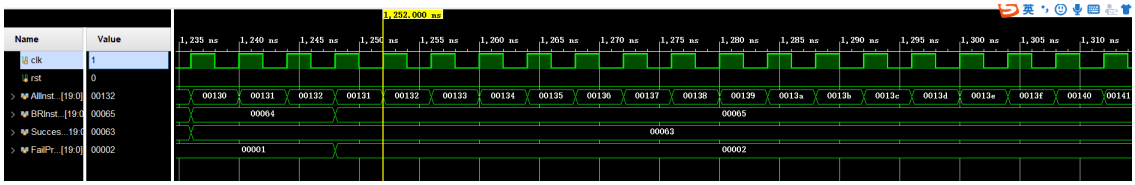
- BTB+Quicksort



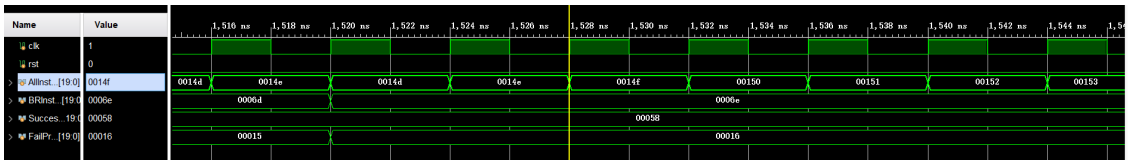
- BTB+MatMul



- BTB+btb.S



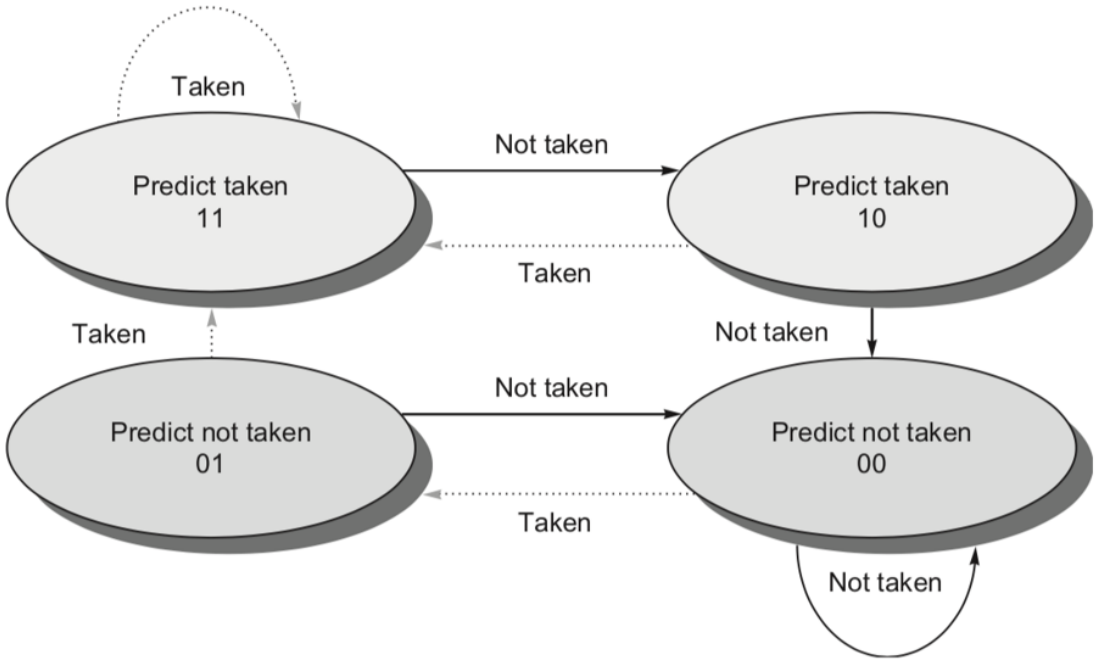
- BTB+bht.S



阶段2

实现思路

- 根据PC的低位查找BHT表，每个项都维护了一个独立的2-bit状态机
- 在EX阶段，BHT表根据实际的跳转结果，更新2-bit的状态机，BTB表则在冲突时更新。状态机如下



- BHT和BTB一样，在IF阶段对当前PC预测其是否跳转。在IF阶段，首先判断当前PC在BTB表中是否跳转，如果跳转，再到BHT表中寻找其是否跳转。只有两者都预测跳转时，才预测当前指令跳转，并将BTB表中的预测跳转地址作为下一条指令的PC地址。特别的，如果BHT表预测跳转，BTB表预测不跳转，或者BHT表预测不跳转，BTB表预测跳转，都不预测当前指令跳转。

BTB.sv

- 在实现上与阶段1略有区别，因为此时决定是否跳转主要取决于BHT表，所以BTB表的作用简化了，只记录那些实际跳转的branch指令的信息，不再需要state位。即若EX阶段发现一个Branch指令实际跳转了，那么将其覆盖写到BTB表里

BHT.sv

- 实现上并不复杂，接口与BTB.sv类似
- 根据实验手册所述，是用PC值的低位去BHT表里查询即可，这里需要注意低2bit需要略去。
- 读BHT表时，若对应表项值为00,01，则预测不跳转；否则预测跳转

```
always @(*)
begin
    ReadPredictTaken = BHTTable[ReadAddr] >= 2'b10;
end
```

- 更新BHT表时，只需要根据当前EX段是否是Branch指令，以及其实际跳转与否来更新即可

  若当前是Branch指令，那么需要更新 （参考实验手册里的状态机）

  - 若实际跳转，那么+1
  - 若实际不跳转，则-1

```
    begin
        // 根据状态图所示，修改BHT表里对应位置的状态即可
        if(bht_w) begin
            if(WriteTaken)
            begin
                if(BHTTable[WriteAddr] != 2'b11)
                    BHTTable[WriteAddr] <= BHTTable[WriteAddr] + 2'b01;
                else
```

```
                    BHTTable[WriteAddr] <= BHTTable[WriteAddr];
            end
            else
            begin
                if(BHTTable[WriteAddr] != 2'b00)
                    BHTTable[WriteAddr] <= BHTTable[WriteAddr] - 2'b01;
                else
                    BHTTable[WriteAddr] <= BHTTable[WriteAddr];
            end
        end
    end
```

NPC_Generator.v

- 思路与阶段一完全一致，不过判断条件有一定变化，因为多了BHT的预测值
  - IF段BHT和BTB同时预测跳转，才实际使用预测的值
  - EX段 没预测或预测不跳转，但实际跳转了，那么需要更新为Branch指令的目的地址
  - EX段 预测跳转，实际不跳转，那么需要更新PC_In为PCE+4

```
always@(*)
begin
//注意这里判断的顺序体现了优先级，EX阶段的优先级高于ID段,具体原因可参考lab1报告的分析
    if (JalrE)
        PC_In <= JalrTarget;
    else if((~PredictedE || PredictedE && ~PredictedTakenE) && BranchE) // 没预测或预测不跳转，但实际跳转了
        PC_In <= BranchTarget;
    else if(~BranchE && PredictedE && PredictedTakenE) // 预测跳转，实际不跳转
        PC_In <= PCE + 4;
    else if(JalD)
        PC_In <= JalTarget;
    else if(PredictedF && PredictedTakenF) // 预测且预测跳转
        PC_In <= PredictedTargetF;
    else
        PC_In <= PCF + 4;
end
```

HazardUnit.v

- 修改的位置与阶段一一致，不同的点也是在于多了一组信号，需要进行flush的情况即：jalrE，或者遇到branch指令时，分支预测与实际执行情况矛盾

```
else if(JalrE || (PredictedE && (BranchE ^ PredictedTakenE)) || (~PredictedE && BranchE))

 {StallF,FlushF,StallD,FlushD,StallE,FlushE,StallM,FlushM,StallW,FlushW} <= 10'b0001010000;
```

IDSegReg.v

- 只需要正常的把Predicted和PredictedTaken信号传递下去即可

ExSegReg.v

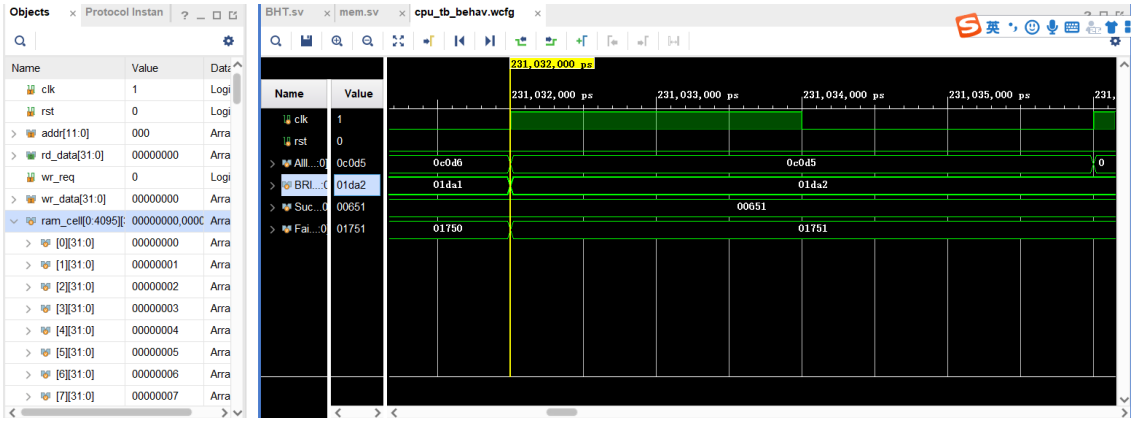
- 只需要正常的把Predicted信号以及PredictedTaken信号传递下去即可

RV32Core.v

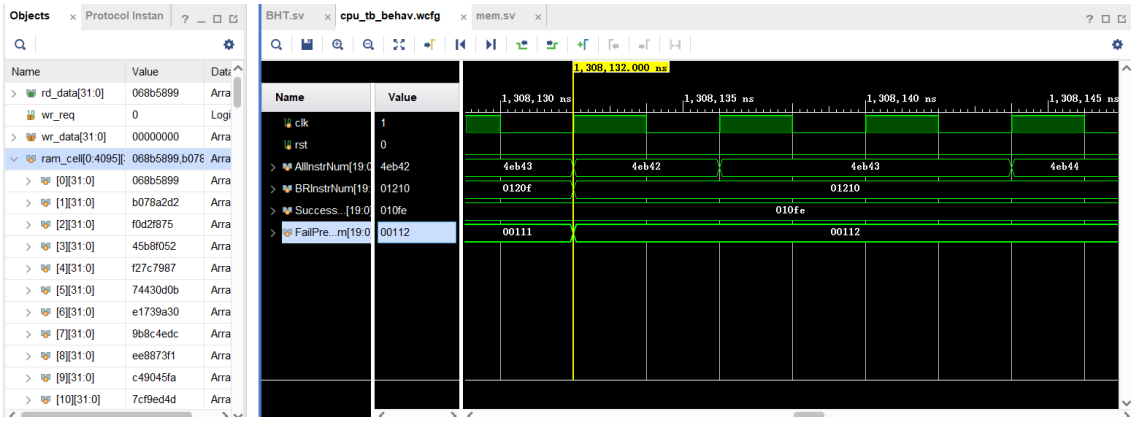
- 只需要是对上述提到修改的模块的接口作一一对接

运行结果

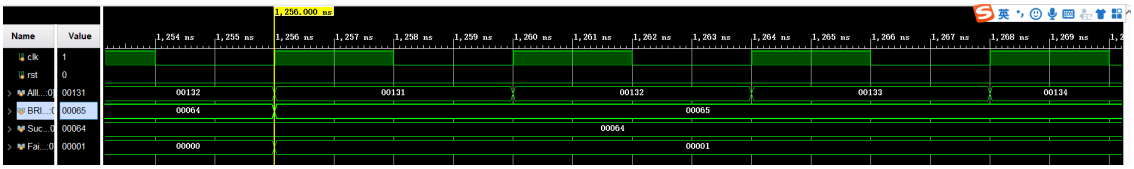
- BHT+QuickSort



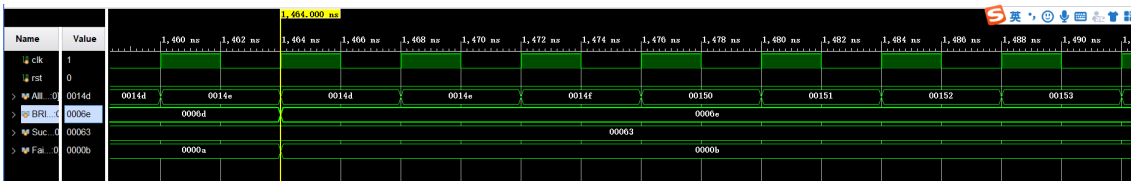
- BHT+MatMul



- BHT+btb.s



- BHT+bht.s



阶段3

补全表格

BTB	BHT	REAL	NPC-PRED	flush	NPC-REAL	BTB-UPADTE
Y	Y	Y	BUF	N	BUF	N
Y	Y	N	BUF	Y	PC_EX+4	N
Y	N	Y	PC_IF+4	Y	BUF	N
Y	N	N	PC_IF+4	N	PC_EX+4	N
N	Y	Y	PC_IF+4	Y	BUF	Y
N	Y	N	PC_IF+4	N	PC_EX+4	N
N	N	Y	PC_IF+4	Y	BUF	Y
N	N	N	PC_IF+4	N	PC_EX+4	N

这里要注意，若EX段发现是一条实际跳转的branch指令，那么会在EX段更新BUF，所以NPC_Real仍取BUF

4个测试样例执行与分析

分析分支收益和分支代价

- 预测正确时收益为2个cycle
- 预测失败是代价为2个cycle

统计未使用分支预测和使用分支预测的总周期数及差值

- 总周期数

	btb.s	bht.s	QuickSort.s	MatMul.s
无分支预测	510	538	67001	354611
BTB	316	386	59518	327565
BHT	314	366	57758	327033

- 差值

	btb.s	bht.s	QuickSort.s	MatMul.s
BTB	194	152	7483	27046
BHT	196	172	9243	27578

统计分支指令数目、动态分支预测正确次数和错误次数

	btb.s	bht.s	QuickSort.s	MatMul.s
分支指令数目	101	110	7586	4624
BTB预测正确次数	99	88	5646	4076
BTB预测错误次数	2	22	1940	548
BHT预测正确次数	100	99	1617	4350
BHT预测错误次数	1	11	5969	274

对比不同策略并分析以上几点的关系

- 使用动态分支预测方法相较于不使用分支预测，周期数都减少了很多，并且BHT减少的要多一些
  - 这是因为程序主要由循环构成，且跳转情况较多，所以使用BTB和BHT都可以获得正收益
  - 分支指令只占指令的一部分，所以优化分支指令只能有限地减少周期数
- 对比BHT和BTB，多数情况下，BHT预测正确的次数都更多一些，说明其效果更好。但是在QuickSort场景下，反而BTB预测的效果更好，这可能是由于程序的循环结构，以及跳转情况的特殊性决定的。

  所以实际使用时，BTB模块和BHT模块的选择需要根据具体应用、具体要求。

总结

- 通过本次实验，完整实现了动态分支预测中的BTB以及BHT策略，并进行了性能测试及相关信息统计，通过实际与理论相结合的方式做了分析，体会了动态分支预测对于程序性能的提升
- 本次实验工作量整体来说不大，但要细心，并且要实现弄清楚BTB和BHT的原理，这样才能较为快速的构建模块

改进意见

暂无