lab4 分支预测

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

实验环境

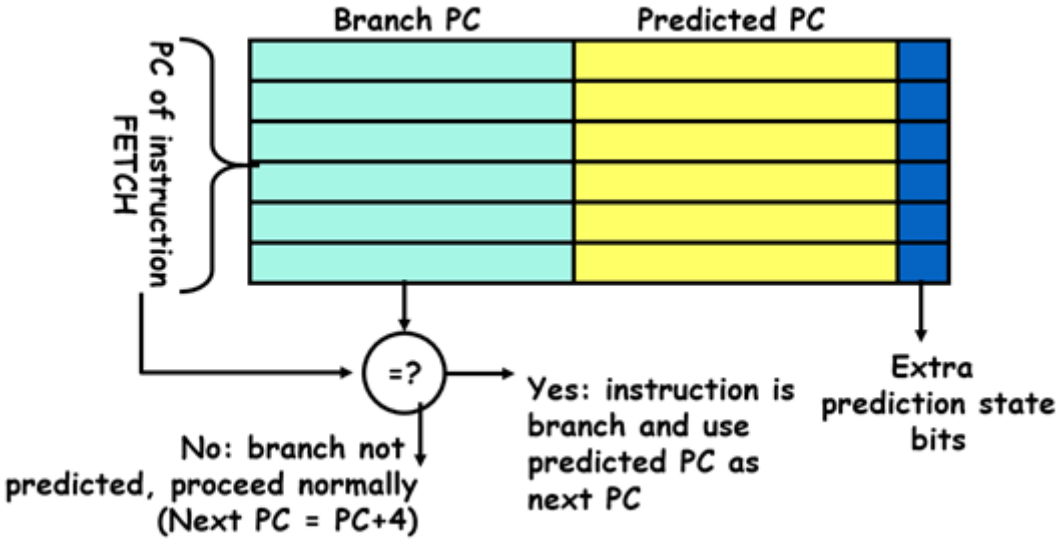
- 编辑器：vscode
- 仿真：vivado 2019.1

实验内容与过程

阶段1

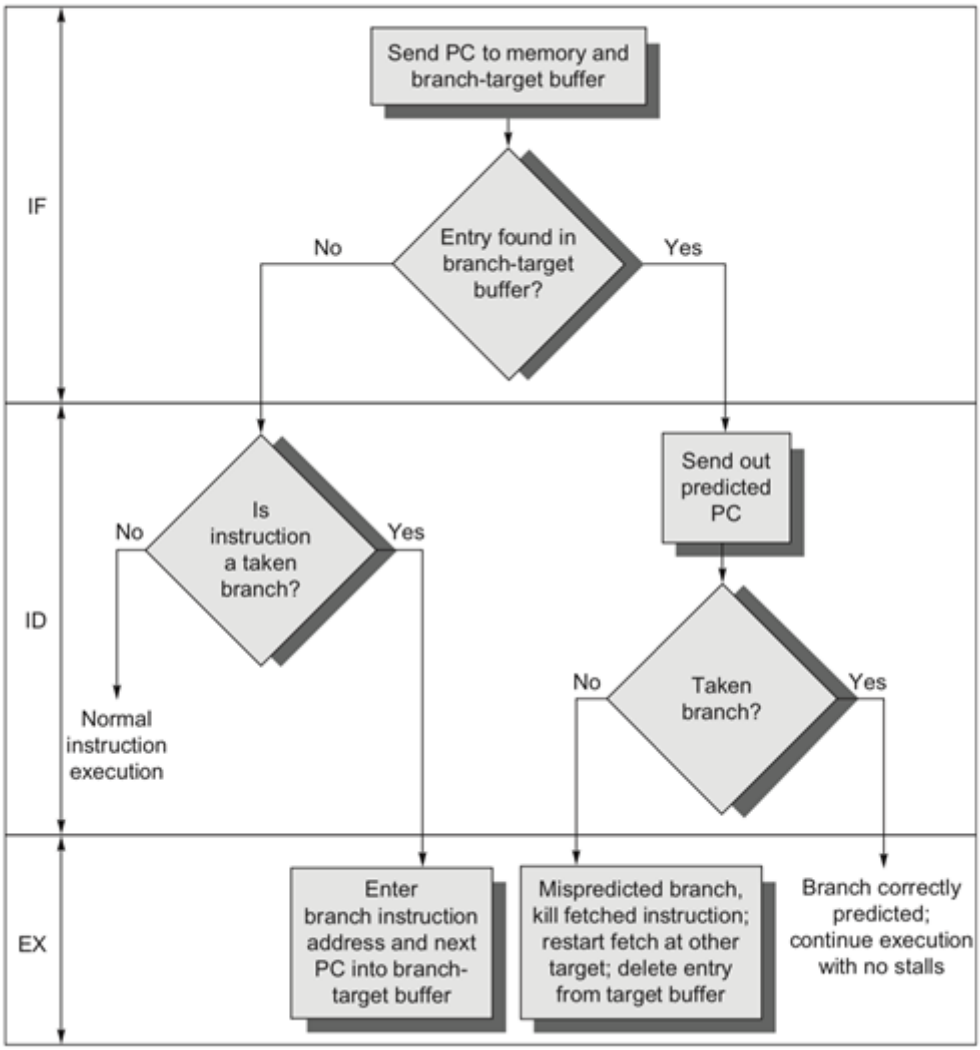
- `BTB.v`

  BTB（Branch Target Buffer）是动态分支预测的一种基本方法。它使用一个Buffer，里面记录了历史指令跳转信息。**对于每一条跳转的Branch指令，它都将其写入buffer，记录其跳转的地址，并有一个标志位标记最近一次执行是否跳转。**这样如果有一条在Buffer里的跳转指令将执行时，可以根据buffer记录的历史跳转信息，预测下一条要执行的指令地址，预测正确的话可以减小分支开销。BTB只使用了**1-bit的历史信息**，也可以视作1-bit BHT。

  在之前实现的lab3 RV32I Core中，下一条PC地址是PC + 4。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，根据其历史跳转记录，确定是否选择predicted PC作为下一PC。**如果当前PC不在BTB表里，但**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。**另外，如果PC在BTB表中，**在EX阶段发现预测的跳转失败，也需要更新BTB表，并flush错误装载的指令。**

状态机如下图



接口设计：

```
module BTB #(
    parameter  BUFFER_ADDR_LEN = 12     //Buffer 的大小
)(
    input  clk, rst,                    //时钟、复位
    input [31:0] PCRead,                //输入PC
    output reg ReadPredict,             //对外输出的信号，为 1 表示 PCRead 是
Branch指令，对应 PCReadPredict 是有效数据
    output reg [31:0] PCReadPredict,    //从 buffer 中读出的预测 PC

    input BTBWrite,                     //写请求
    input [31:0] PCWrite,               //要写入的分支 Branch
    input [31:0] PCWritePredict,        //要写入的PredictPC
    input StateBitWritePredict          //要写入的预测状态位
);
```

标签对应的地址长度和 Buffer 大小如下

```
localparam TAG_ADDR_LEN = 32 - BUFFER_ADDR_LEN - 2; //计算tag的数据位宽
localparam BUFFER_SIZE = 1 << BUFFER_ADDR_LEN;      //计算buffer的大小
```

直接映射，故将 32 bits PC 地址差分成如下三部分，最后两位始终为0

```
wire [BUFFER_ADDR_LEN - 1 : 0] ReadBufferAddr;
wire [   TAG_ADDR_LEN - 1 : 0] ReadTagAddr;
wire [                  1 : 0] ReadWordAddr;

wire [BUFFER_ADDR_LEN - 1 : 0] WriteBufferAddr;
wire [   TAG_ADDR_LEN - 1 : 0] WriteTagAddr;
wire [                  1 : 0] WriteWordAddr;

assign {ReadTagAddr, ReadBufferAddr, ReadWordAddr} = PCRead; //拆分 32bits
PCRead
assign {WriteTagAddr, WriteBufferAddr, WriteWordAddr} = PCWrite; //拆分
32bits PCWrite
```

设置数组 `PCTag`、`PCPredict`、`StateBitPredict`，分别对应标签、预测的 PC 值以及状态位

```
reg [TAG_ADDR_LEN - 1 : 0] PCTag            [0 : BUFFER_SIZE - 1]; //分支PC
TAG
reg [              31 : 0] PCPredict        [0 : BUFFER_SIZE - 1]; //预测PC
reg                        StateBitPredict  [0 : BUFFER_SIZE - 1]; //预测状态
位
```

读取 Buffer，判断输入的 PC 是否在 Buffer 中命中。如果命中且预测状态位为1，则预测跳转，即 `ReadPredict = 1'b1`；否则 `ReadPredict = 1'b0`。

```
// 读取 buffer
always @(*)
begin
    //判断输入的 PC 是否在 buffer 中命中
    if(PCTag[ReadBufferAddr] == ReadTagAddr &&
StateBitPredict[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;
    PCReadPredict = PCPredict[ReadBufferAddr];
end
```

在 EX 段中，如果预测的跳转和实际情况不一致，则需要更新 BTB

```
// 写入 buffer
integer i;
always @(posedge clk or posedge rst)
begin
    if(rst)
    begin
        for(i = 0; i < BUFFER_SIZE; i = i + 1)
        begin
            PCTag[i] = 0;
            PCPredict[i] = 0;
            StateBitPredict[i] = 1'b0;
        end
        PCReadPredict = 0;
        ReadPredict = 1'b0;
    end
    else
```

```
    begin
        if(BTBWrite)
        begin
            PCTag[WriteBufferAddr] <= WriteTagAddr;
            PCPredict[WriteBufferAddr] <= PCWritePredict;
            StateBitPredict[WriteBufferAddr] <= StateBitWritePredict;
        end
    end
end
```

- `IFSegReg.v`、`IDSegReg.v`需要作少量修改，传递`BRPredicted`的信号即可
- `NPC_Generator.v`：
  - 在 EX 段，如果预测不跳转但实际跳转，则要将 PC 修改为跳转的地址
  - 在 EX 段，如果预测跳转实际不跳转，则要将 PC 修改为 PCE + 4
  - 在 IF 段，如果预测跳转，则选择预测的 PC 作为下一个 PC

```
    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
        if (JalrE)
            PC_In <= JalrTarget;
        else if(BranchE && ~BRPredictedE) // 预测不跳转，实际跳转
            PC_In <= BranchTarget;
        else if(~BranchE && BRPredictedE) // 预测跳转，实际不跳转
            PC_In <= PCE + 4;
        else if(JalD)
            PC_In <= JalTarget;
        else if(BRPredictedF) // 预测即跳转
            PC_In <= BRPredictedTargetF;
        else
            PC_In <= PCF + 4;
    end
```

- `HarzardUnit.v`：如果 EX 段发现预测情况与实际情况不一致，则需要冲刷掉后面两条指令

```
else if ((BranchE ^ BRPredictedE) | JalrE)
        begin
            StallF <= 1'b0;
            FlushF <= 1'b0;
            StallD <= 1'b0;
            FlushD <= 1'b1;
            StallE <= 1'b0;
            FlushE <= 1'b1;
            StallM <= 1'b0;
            FlushM <= 1'b0;
            StallW <= 1'b0;
            FlushW <= 1'b0;
        end
```

- `RV32Core.v`主要是添加对应的数据通路、变量等等，主要是修改`BTB`、`NPC_Generator`、`IDSegReg`、`EXSegReg`、`HarzardUnit`，以及增加一个计数器。核心代码如下

```
        BTB BTBInst(
        .clk(~CPU_CLK),
        .rst(CPU_RST),
        .PCRead(PCF),//输入PC
        .ReadPredict(BRPredictedF),//对外输出的信号，为 1 表示 PCRead 是Branch指
令，对应 PCReadPredict 是有效数据
        .PCReadPredict(BRPredictedPCF),//从 buffer 中读出的预测 PC
        .BTBWrite(BRPredictedE ^ BranchE),//写buffer，预测和实际不同
        .PCWrite(PCE),//要写入的分支 Branch
        .PCWritePredict(BrNPC),//要写入的PredictPC
        .StateBitWritePredict(BranchE) //要写入的预测状态位
    );
    // 计数器
    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(BRPredictedE ^ BranchE)
                    FailPredictNum <= FailPredictNum + 1;
                else
                    SuccessPredictNum <= SuccessPredictNum + 1;
            end
        end
    end
```

阶段2

- `BHT.sv`

  BHT（Branch History Table）是动态分支预测的另一种基本策略。类似BTB，它也维护了一个N * 2的cache作为buffer。其中，N是BHT表的项数（**一般取4096项**），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。

  BHT和BTB一样，在IF阶段对当前PC预测其是否跳转。相较于BTB，BHT的预测准确度更高，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。

  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。

  接口如下

```
module BHT #(
    parameter  TABLE_ADDR_LEN = 12 //BHTTable 大小，与BTB中的 BUFFER_ADDR_LEN
保持一致
)(
    input  clk, rst,
    input [31:0] PCRead,                //输入PC
    output reg ReadPredictTaken,        //输出信号，为 1 表示预测 PCRead 跳转
    input BHTWrite,                     //写请求
    input [31:0] PCWrite,               //要更新的 Branch
    input WriteTaken                  //要更新的 Branch 实际是否跳转
);
```

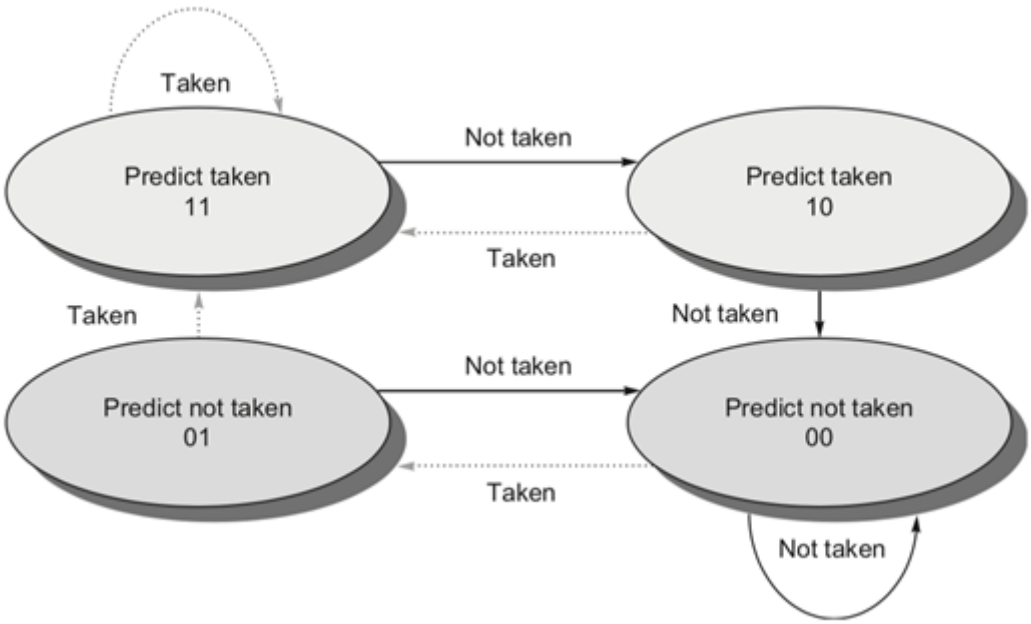
在阶段 1 的基础上增加一个 2bits 的 BHT

```
reg [1 : 0] BHTTable [0 : TABLE_SIZE - 1];
```

只有 BTB 和 BHT 都预测跳转时才命中

```
// 读取 buffer：0，1预测不跳转，2，3预测跳转
always @(*)
begin
    ReadPredictTaken = BHTTable[ReadTableAddr] >= 2'b10;
end
```

EX 段需要更新 BHT 的状态，状态机如下



对应代码如下

```
// 写入buffer
always @(posedge clk or posedge rst)
begin
    if(rst)
    begin
        for(i= 0; i < TABLE_SIZE; i = i + 1)
        begin
```

```
            BHTTable[i] = 2'b00;
        end
        ReadPredictTaken = 1'b0;
    end
    else
    begin
        // 按照给出的状态转换图更新PC对应表项的状态
        if(BHTWrite) begin
            if(WriteTaken)
            begin
                if(BHTTable[WriteTableAddr] != 2'b11)
                    BHTTable[WriteTableAddr] <= BHTTable[WriteTableAddr] +
2'b01;
                else
                    BHTTable[WriteTableAddr] <= BHTTable[WriteTableAddr];
            end
            else
            begin
                if(BHTTable[WriteTableAddr] != 2'b00)
                    BHTTable[WriteTableAddr] <= BHTTable[WriteTableAddr] -
2'b01;
                else
                    BHTTable[WriteTableAddr] <= BHTTable[WriteTableAddr];
            end
        end
    end
end
```

- `NPC_Generator.v`：
  - 在 EX 段，如果没预测或者预测不跳转但实际跳转，则要将 PC 修改为跳转的地址
  - 在 EX 段，如果预测跳转实际不跳转，则要将 PC 修改为 PCE + 4
  - 在 IF 段，如果预测且预测跳转，则选择预测的 PC 作为下一个 PC

```
    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
        if (JalrE)
            PC_In <= JalrTarget;
        // 没预测或预测不跳转，但实际跳转了
        else if(BranchE && (~BRPredictedE || BRPredictedE &&
~BRPredictedTakenE))
            PC_In <= BranchTarget;
        // 预测跳转，但实际不跳转
        else if(~BranchE && BRPredictedE && BRPredictedTakenE) // 预测跳
转，实际不跳转
            PC_In <= PCE + 4;
        else if(JalD)
            PC_In <= JalTarget;
        else if(BRPredictedF && BRPredictedTakenF) // 预测且预测跳转
            PC_In <= BRPredictedTargetF;
        else
            PC_In <= PCF + 4;
    end
```

- `HarzardUnit.v`：如果进行预测但预测结果和实际跳转结果不一致，或者没有预测但实际跳转了，则需要 flush 掉后面两条指令

```
        //进行预测但预测结果和实际跳转结果不一致
        //没有预测但实际跳转了
        else if (JalrE || (BRPredictedE && (BranchE ^ BRPredictedTakenE)) ||
(~BRPredictedE && BranchE))
        begin
            StallF <= 1'b0;
            FlushF <= 1'b0;
            StallD <= 1'b0;
            FlushD <= 1'b1;
            StallE <= 1'b0;
            FlushE <= 1'b1;
            StallM <= 1'b0;
            FlushM <= 1'b0;
            StallW <= 1'b0;
            FlushW <= 1'b0;
        end
```

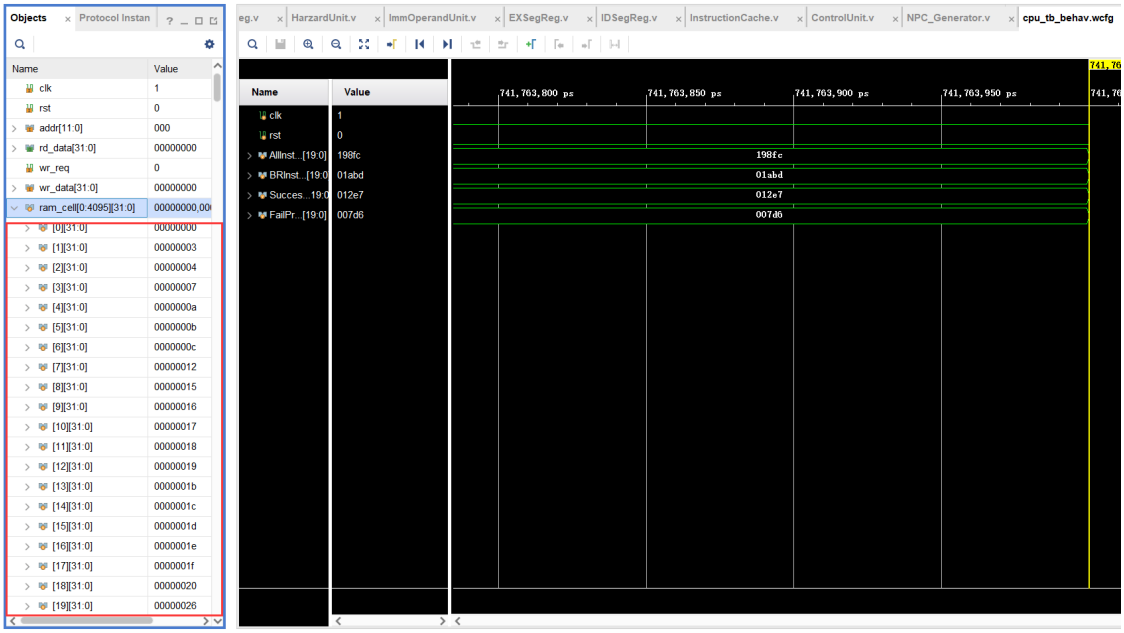
- `RV32Core.v`：在阶段 1 的基础上添加了对 BHT 模块的调用

```
    BHT BHTInst(
        .clk(~CPU_CLK),
        .rst(CPU_RST),
        .PCRead(PCF),//输入PC
        .ReadPredictTaken(BRPredictedTakenF),//输出信号，为 1 表示预测 PCRead 跳
转
        .BHTWrite(BranchTypeE != 3'b000),
        .PCWrite(PCE),
        .WriteTaken(BranchE)
    );
```

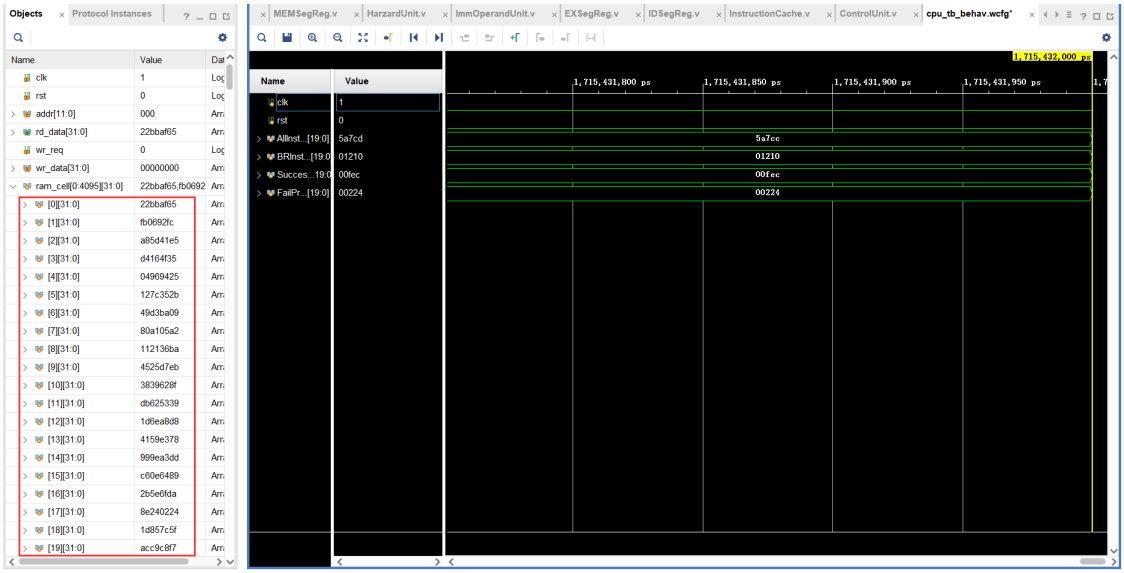
实验结果与分析

结果截图

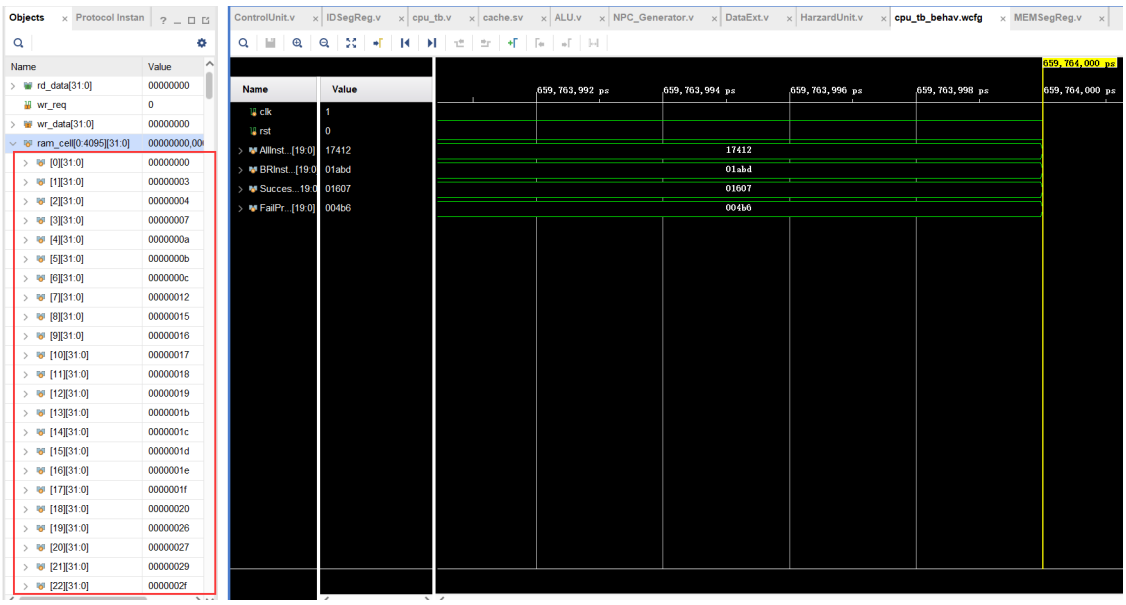
- BTB + QuickSort



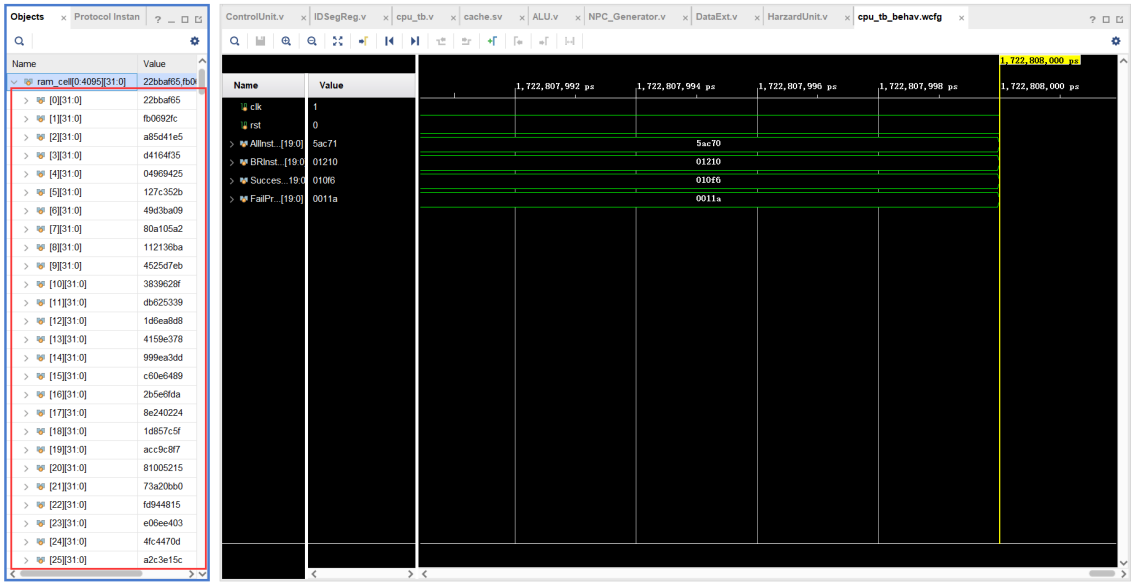
- BTB + MatMul



- BHT + QuickSort



- BHT + MatMul



BHT（Branch History Table）

BTB	BHT	REAL	NPC_PRED	flush	NPC_REAL	BTB update
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

（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）

结果分析

运行**btb.s、bht.s、QuickSort.s、MatMul.s**四个测试样例并分析

分析分支收益和分支代价

预测正确时收益为 2 个周期

预测失败时代价为 2 个周期

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
未使用分支预测	510	538	67001	354611
BTB	316	386	67401	346991
BHT	320	388	65891	347001

- 总周期数差值

	btb.s	bht.s	QuickSort.s	MatMul.s
BTB	194	152	-400	7620
BHT	190	150	1110	7610

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	16300	4612
BTB预测正确次数	99	88	12130	4070
BTB预测错误次数	2	22	4170	542
BHT预测正确次数	98	95	15222	4072
BHT预测错误次数	3	15	1078	540

对比不同策略并分析以上几点的关系

1. 使用动态分支预测相较于不使用分支预测，总周期数少了很多
    - 对于 btb.s、bht.s、MatMul.s 来说，程序主要由循环构成，且跳转的情况较多。所以使用动态分支预测（BTB 和 BHT）都可以获得正收益
    - 但是对于 QuickSort.s 来说，由于其内部循环较为复杂，所以使用简单的 BTB 预测反而没有全部预测不跳转效果好，从而出现了负优化。但是使用更加负责的 BHT 可以带来一定的正优化
2. 分支指令只是所有指令中较少的一部分，所有优化分支指令只能有限地减少周期数。每预测成功一次都可以带来两个周期的优化
3. 多数情况下 BTB 和 BHT 优化效果差不多，但是 BTB 显然更加简单，所以使用 BTB 就可以了。但是对于较为复杂的情况， BTB 效果不如 BHT，甚至不如不使用动态分支预测

实验总结

在本次实验中，我通过实现动态分支预测中的 BTB 以及 BHT，了解了动态分支预测的具体底层实现。在结果分析中通过统计分支指令数目、动态分支预测正确次数和错误次数，并和不使用分支预测的结果作对比，体会到了动态分支预测对程序性能的提升。

这次实验也主要是在前面实验的基础上增加 BTB 模块和 BHT 模块，并修改少量的其他模块，整体工作量不是很大，但仍有许多细节需要注意，否则很容易出 Bug。总体来说，收获颇丰。