计算机体系结构Lab4实验报告

廖洲洲 PB17081504

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

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	br_target	Y
N	Y	N	PC_IF+4	N	PC_IF+4	N
N	N	Y	PC_IF+4	Y	br_target	Y
N	N	N	PC_IF+4	N	PC_IF+4	N

BTB

- BTB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。
- 当在BTB中未检查到匹配项，或匹配但状态位为0，则下一条指令为PC+4，但在EX段发现是需要跳转的Branch指令，则在EX阶段更新BTB表，（此时Br=1，跳转，同时flush)；是不需要跳转的Branch指令和其它指令则不用做其它事。
- 当在BTB中检查到匹配项，并且状态位为1，则使用Predicted PC作为下一PC。如果在EX阶段发现其跳转成功，不需要做任何事(令Br=0即可）；如果跳转失败，更新BTB表，并flush错误装载的指令，执行PC+4（在EX阶段能得到之前的PC，故再设置一个PC_4_Old选择之前的PC+4）。
- buffer 放在取指阶段，buffer内容读取一个周期内可以完成。
- BTB的命中：当前指令的地位用于寻址，对比指令的高位和buffer中是否相等并且有效位为1，表示命中，则下一条指令的地址不是pc+4，而是buffer中的内容。
- 主要代码(添加BtbBuffer模块，修改NPCGenerator,增加BranchSwitch，增加分支数目统计)

```
module BtbBuffer #(
```

```
    parameter SET_ADDR_LEN = 7, //直接映射，组地址长度为7，说明有2^7=128组
    parameter TAG_PC_LEN = 2,   //PCtag长度
    parameter STATE_LEN = 1     //标志最后一次执行是否跳转
 )(
    input clk,rst,
    input [31:0] PC_IF,         //输入IF阶段产生的PC，用于比较是否Buffer中是否有对
应项
    output reg isTakenBr_BTB,       //根据PC_IF查找Buffer，预测分支是否跳转
    output reg [31:0] predictedPC, //若预测分支跳转，输出预测跳转地址

    input wr_req,               //更新Buffer请求
    input [31:0] PC_EX,         //输入EX阶段的PC，用于更新Buffer表
    input [31:0] PC_Branch,     //输入EX阶段Branch指令对应的分支地址，用于更新
Buffer表
    input isTakenBr_Ex          //更新历史跳转信息
 );

 //localparam LINE_SIZE = 1 << LINE_ADDR_LEN;
localparam SET_SIZE = 1 << SET_ADDR_LEN;
localparam UNSED_ADDR_LEN = 32 - TAG_PC_LEN - SET_ADDR_LEN - 2;

reg [31 : 0] predictedPC_buffer[SET_SIZE];
reg [TAG_PC_LEN-1 : 0] tagPC_buffer[SET_SIZE];
reg state_buffer [SET_SIZE];

wire [2-1:0] word_PC_IF;
wire [SET_ADDR_LEN -1 : 0] set_PC_IF;
wire [TAG_PC_LEN - 1 : 0] tag_PC_IF;
wire [UNSED_ADDR_LEN - 1 : 0] unused_PC_IF;


wire [2-1:0] word_PC_EX;
wire [SET_ADDR_LEN -1 : 0] set_PC_EX;
wire [TAG_PC_LEN - 1 : 0] tag_PC_EX;
wire [UNSED_ADDR_LEN - 1 : 0] unused_PC_EX;
//分割PC地址
assign {unused_PC_IF, tag_PC_IF, set_PC_IF, word_PC_IF} = PC_IF;
assign {unused_PC_EX, tag_PC_EX, set_PC_EX, word_PC_EX} = PC_EX;

always@(*)begin
    if(tagPC_buffer[set_PC_IF] == tag_PC_IF && state_buffer[set_PC_IF] ==
1)begin
        isTakenBr_BTB = 1'b1;
        predictedPC = predictedPC_buffer[set_PC_IF];
    end
    else begin
        isTakenBr_BTB = 1'b0;
        predictedPC = 0;
    end
end

always @ (posedge clk or posedge rst) begin
    if(rst) begin
        for(integer i = 0; i < SET_SIZE; i++)begin
            predictedPC_buffer[i] <= 0;
            tagPC_buffer[i] <= 0;
            state_buffer[i] <= 0;
        end
```

```
    end
    else begin
        if(wr_req) begin
            tagPC_buffer[set_PC_EX] <= tag_PC_EX;
            state_buffer[set_PC_EX] <= isTakenBr_Ex;
            predictedPC_buffer[set_PC_EX] <= PC_Branch;
        end
    end
end

 endmodule
```

```
module BranchSwitch(
    input wire realBr,
    input wire isTakenBr_BTB_EX,
    output reg br,
    output reg br_target_Or_PC4,
    output reg wr_req_btb
);

always@(*)
begin
    if(realBr != isTakenBr_BTB_EX)begin
       br <= 1;
       wr_req_btb <= 1;
       if(realBr)begin
           br_target_Or_PC4 <= 1;
       end else begin
           br_target_Or_PC4 <= 0;
       end
    end
    else begin
        br <= 0;
        br_target_Or_PC4 <= 0;
        wr_req_btb <= 0;
    end
end
endmodule
```

BHT

- 维护了一个N * 2的cache作为buffer, 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IF阶段，首先判断当前PC在BTB表中是否命中，如果命中，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。此时，BTB就相当于一个Buffer保存每条分支指令的分支地址，而是否分支需要由BHT来决定。因此只有再BHT未命中而指令是Br且分支时需要更新
- 主要代码(在BTB的基础上增加BhtBuffer,修改BtbBuffer,修改BranchSwitch，增加分支数目统计)

```
module BhtBuffer #(

```

```
    //parameter LINE_ADDR_LEN = 2,// 每一路地址长度，决定了每一路具有4word，即预测的PC为32位
    parameter SET_ADDR_LEN = 12 //直接映射，组地址长度为12，说明有2^12=4096组
    //parameter TAG_PC_LEN = 2,    //PCtag长度
    //parameter STATE_LEN = 1      //标志最后一次执行是否跳转
 )(
    input clk,rst,
    input [31:0] PC_IF,         //输入IF阶段产生的PC，用于比较是否Buffer中是否有对应项
    input isHit_BTB,             //输入BTB是否命中
    output reg isTakenBr_Bht,       //根据PC_IF查找Buffer，预测分支是否跳转

    input wr_req,               //更新Bht_Buffer请求
    input [31:0] PC_EX,         //输入EX阶段的PC，用于更新Buffer表
    input isTakenBr_Ex          //根据是否真实跳转更新Bth表
 );

 //localparam LINE_SIZE = 1 << LINE_ADDR_LEN;
localparam SET_SIZE = 1 << SET_ADDR_LEN;
localparam UNSED_ADDR_LEN = 32 - SET_ADDR_LEN - 2;


reg [1:0] state_buffer [SET_SIZE];

wire [2-1:0] word_PC_IF;
wire [SET_ADDR_LEN -1 : 0] set_PC_IF;
wire [UNSED_ADDR_LEN - 1 : 0] unused_PC_IF;


wire [2-1:0] word_PC_EX;
wire [SET_ADDR_LEN -1 : 0] set_PC_EX;
wire [UNSED_ADDR_LEN - 1 : 0] unused_PC_EX;
//分割PC地址
assign {unused_PC_IF, set_PC_IF, word_PC_IF} = PC_IF;
assign {unused_PC_EX, set_PC_EX, word_PC_EX} = PC_EX;

always@(*)begin
    if(isHit_BTB && state_buffer[set_PC_IF][1] == 1'b1 ) begin
        isTakenBr_Bht = 1'b1;
    end
    else begin
        isTakenBr_Bht = 1'b0;
    end
end

always @ (posedge clk or posedge rst) begin
    if(rst) begin
        for(integer i = 0; i < SET_SIZE; i++)begin
            state_buffer[i] <= 2'b11;
        end
    end
    else begin
        if(wr_req) begin
            case(state_buffer[set_PC_EX])
            2'b11:  begin
                if(!isTakenBr_Ex)begin
                    state_buffer[set_PC_EX] <= 2'b10;
                end
```

```
            end
            2'b10:  begin
                if(isTakenBr_Ex)begin
                    state_buffer[set_PC_EX] <= 2'b11;
                end else begin
                    state_buffer[set_PC_EX] <= 2'b00;
                end
            end
            2'b00:begin
                if(isTakenBr_Ex) begin
                    state_buffer[set_PC_EX] <= 2'b01;
                end
            end
            2'b01:begin
                if(isTakenBr_Ex) begin
                    state_buffer[set_PC_EX] <= 2'b11;
                end else begin
                    state_buffer[set_PC_EX] <= 2'b00;
                end
            end
            endcase
        end
    end
end

endmodule
```

```
`include "Parameters.v"
module BranchSwitch(
    input wire realBr,
    input wire isHit_BTB,
    input wire [2:0] br_type,
    input wire isTakenBr_BHT_EX,

    output reg br,
    output reg br_target_Or_PC4,
    output reg wr_req_btb,
    output reg wr_req_bht
);

always@(*)
begin
    if(realBr != isTakenBr_BHT_EX)begin
       br <= 1;
       if(realBr)begin
           br_target_Or_PC4 <= 1;
       end else begin
           br_target_Or_PC4 <= 0;
       end
    end
    else begin
        br <= 0;
        br_target_Or_PC4 <= 0;
    end
    if(br_type != `NOBRANCH) begin
        wr_req_bht <= 1'b1;
    end else begin
```

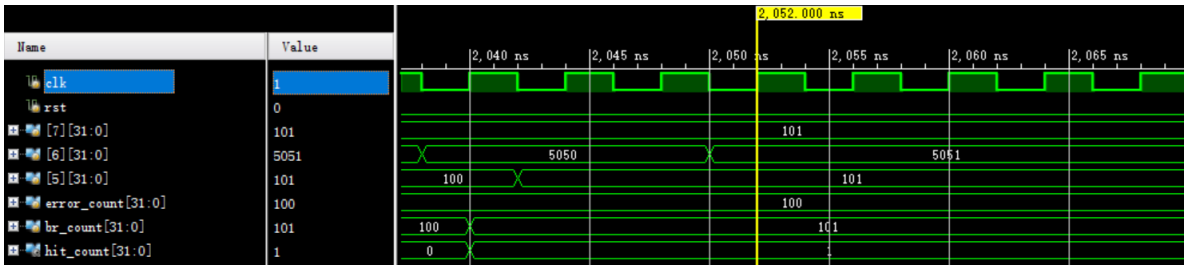
```
            wr_req_bht <= 1'b0;
        end
        if(isHit_BTB == 1'b0 && realBr == 1'b1) begin
            wr_req_btb <= 1'b1;
        end else begin
            wr_req_btb <= 1'b0;
        end
    end

    endmodule
```

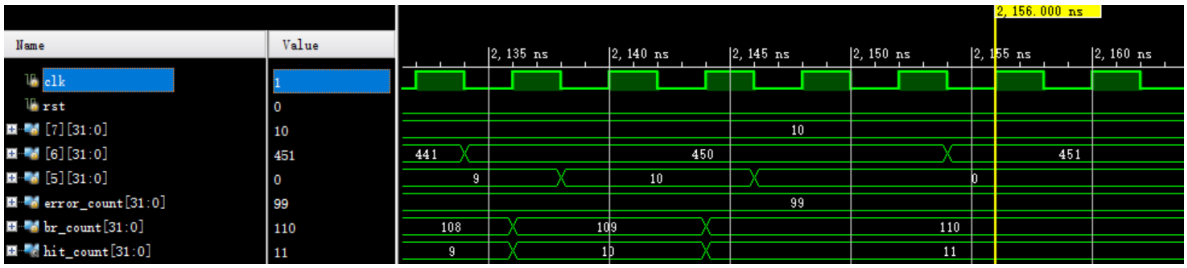
实验结果

无分支预测CPU

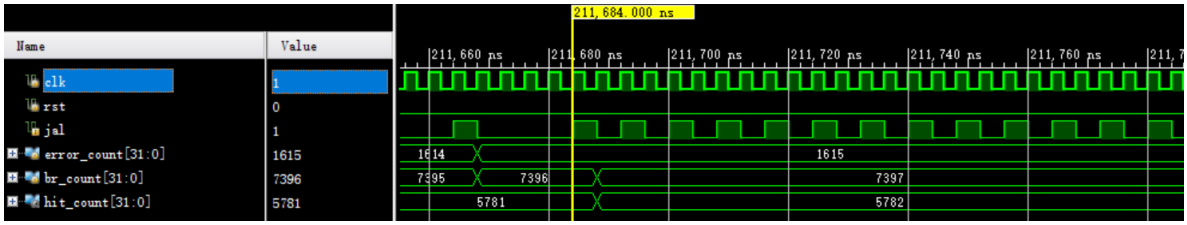
btb.s



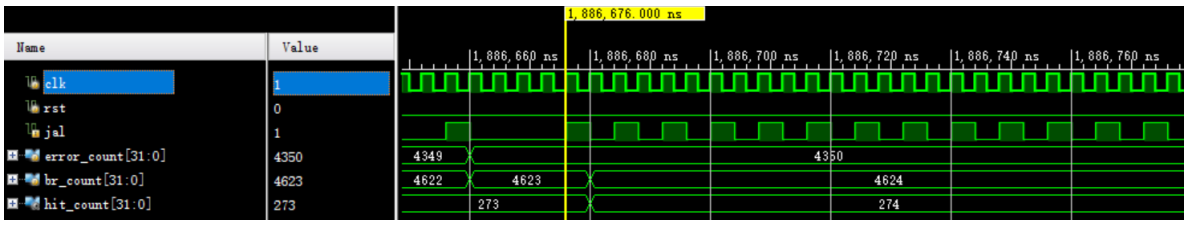
bht.s



QuickSort.s 256

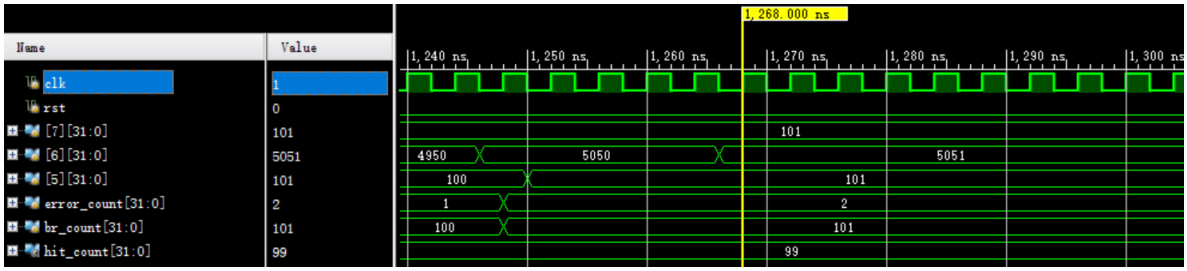


MatMul.s 16*16

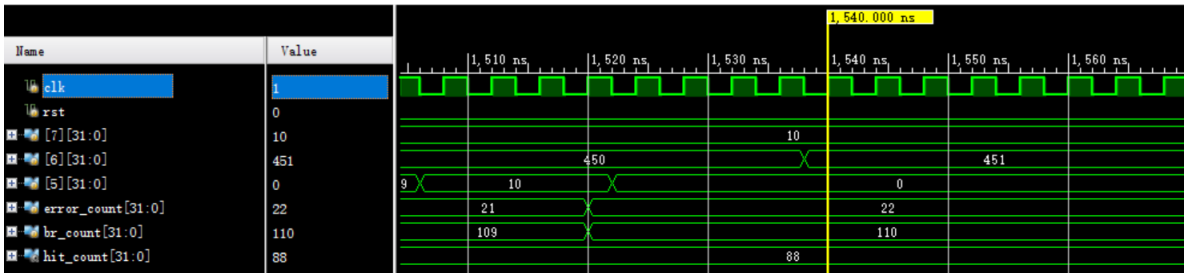


BTB

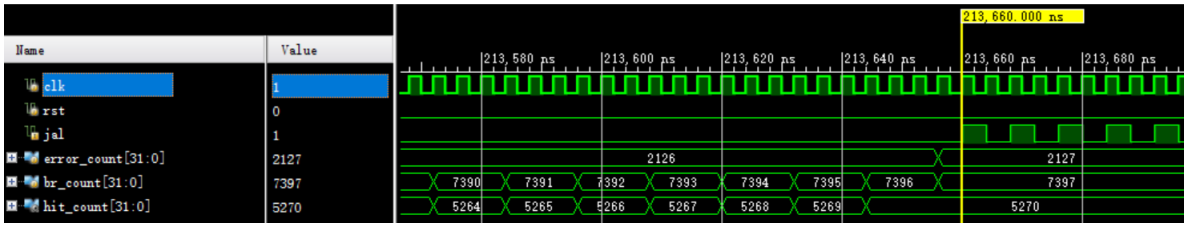
btb.s



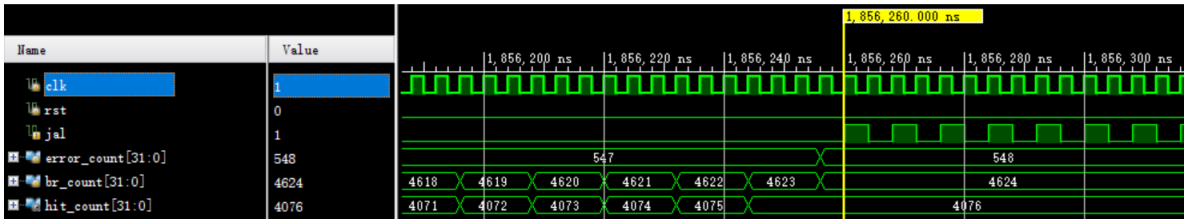
bht.s



QuickSort.s 256

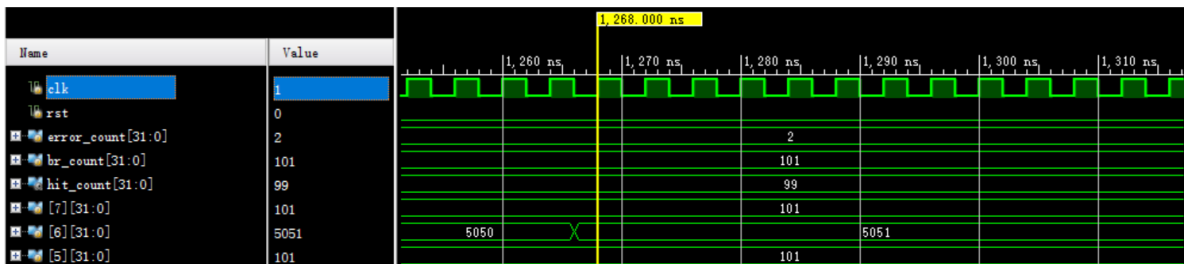


MatMul.s 16*16

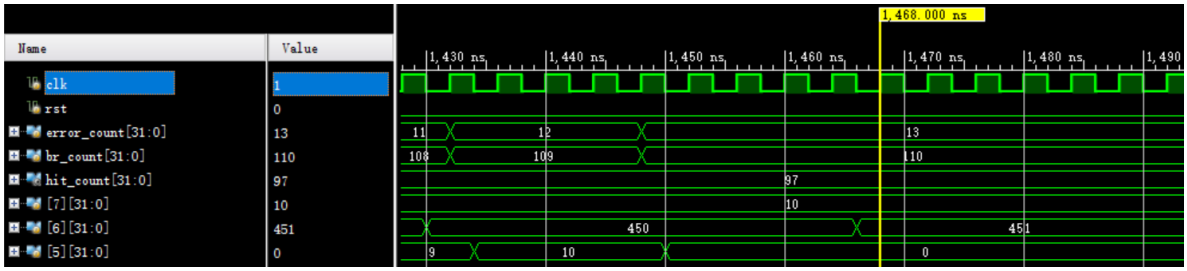


BHT

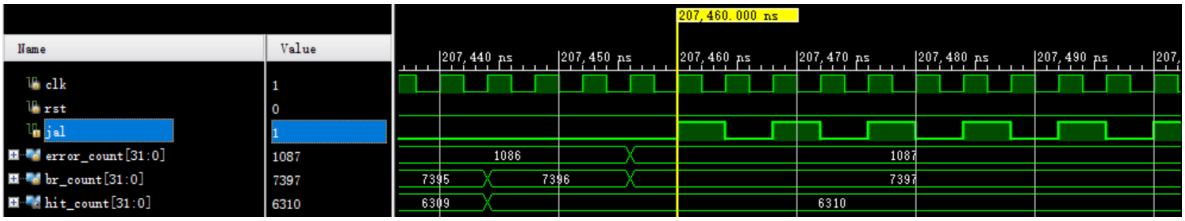
btb.s



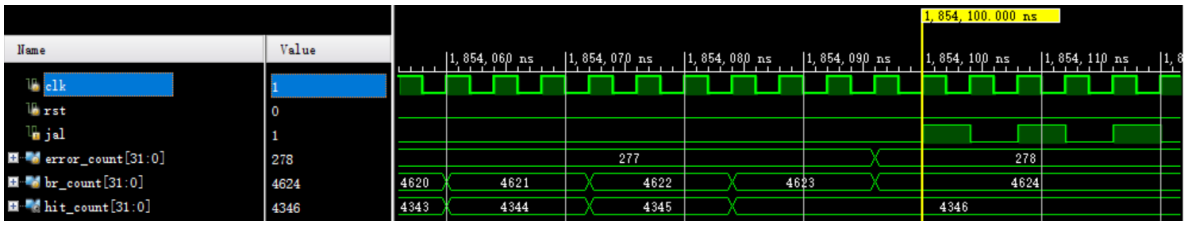
bht.s



QuickSort.s 256



MatMul.s 16*16



实验分析

分析分支收益和分支代价

Lab3CPU

相当于预测不分支CPU，对于Br指令的下一条指令总是选择PC+4，当Br指令确实需要跳转时flush流水线，代价为2个周期。因此当Br是需要跳转的，其CPI相当为3，不需要跳转的则为1。

BTB

根据该条Br指令的上一次的跳转信息预测跳转，当预测错误也是损失2个周期。对于循环执行某条总是跳转的分支指令，其第一次执行时（预测不跳转，真正跳转）和最后一次（预测跳转，真正不跳转）预测错误。其预测错误指令CPI为3，预测成功CPI为1。

BHT

BHT根据一对比特信息预测分支跳转，其预测错误的损失也是2个周期。其对严重偏向跳转或严重偏向不跳转的分支指令性能更佳。其预测错误指令CPI为3，预测成功CPI为1。

使用btb.s进行理论分析：

btb.s中有一个相当于for循环的代码，有1条执行101次的Br指令，前100次执行跳转，第101次执行不跳转。

对于Lab3CPU，最后一次不跳转预测成功，前100次跳转预测不成功。故总周期数
3+101x2+100x3+1+1+4=511周期

对于BTB，第一次跳转和最后一次预测不成功，其它跳转预测成功。故总周期数
3+101x2+2x3+99x1+1+4=315周期

对于BTB，第一次跳转（因为BTB表未命中）和最后一次预测不成功（因为BHT预测跳转），其它跳转预测成功。故总周期数3+101x2+2x3+99x1+1+4=315周期

可以发现理论分析和下面的实验分析一致

统计未使用分支预测和使用分支预测的总周期数及差值

根据测试终止时间计算总周期数和差值，时间减去8ns的RST重置时间，CPU每周期4ns

btb.s

	总时间/ns	总周期	相对Lab3CPU周期差值
Lab3CPU	2052-8=2044	511	0
BTB	1268-8=1260	315	-196
BHT	1268-8=1260	315	-196

bht.s

	总时间/ns	总周期	相对Lab3CPU周期差值
Lab3CPU	2156-8=2148	537	0
BTB	1540-8=1532	383	-154
BHT	1468-8=1460	365	-172

Quisksort.s 256

	总时间/ns	总周期	相对Lab3CPU周期差值
Lab3CPU	211684-8=211676	52919	0
BTB	213660-8=213652	53413	+494
BHT	207460-8=207452	51865	-1054

MatMul.s 16*16

	总时间/ns	总周期	相对Lab3CPU差值
Lab3CPU	1886676-8=1886668	471667	0
BTB	1856260-8=1856252	464063	-7604
BHT	1854100-8=1854092	463523	-8144

一般情况下，BTB和BHT能减少CPU运行周期，且BHT减少得更多

统计分支指令数目、动态分支预测正确次数和错误次数

只统计Br指令

btb.s

	总次数	正确次数	错误次数	正确率
Lab3CPU	101	1	100	0.0099
BTB	101	99	2	0.9802
BHT	101	99	2	0.9802

bht.s

	总次数	正确次数	错误次数	正确率
Lab3CPU	110	11	99	0.1
BTB	110	88	22	0.8
BHT	110	97	13	0.8818

Quisksort.s 256

	总次数	正确次数	错误次数	正确率
Lab3CPU	7397	5782	1615	0.7817
BTB	7397	5270	2127	0.7124
BHT	7397	6310	1087	0.8530

MatMul.s 16*16

	总次数	正确次数	错误次数	正确率
Lab3CPU	4624	274	4350	0.0593
BTB	4624	4076	548	0.8815
BHT	4624	4346	278	0.9399

一般情况下，BTB和BHT的正确预测率都很高，而BHT的正确率会更高些

对比不同策略并分析以上几点的关系

1. Lab3CPU相当于使用预测未选中机制，BTB可以视作1-bit BHT。
2. 使用了BTB和BHT的CPU一般情况下性能相对于Lab3CPU有所提升。
3. 对于简单的for循环（比如前n-1次跳转，第n次不跳转），BTB和BHT均能大幅提升正确预测率。而Lab3CPU对于前n-1次跳转，第n次不跳转型的代码正确率较低。
4. 对于快速排序，由于其是否分支由随机生成的数据的大小决定，故是否分支也是随机的。因此对于依靠上一次跳转信息的BTB来说，其正确预测有可能会低于Lab3CPU，本次生成的数据就造成了这样的结果。
5. 对于矩阵乘法，由于其大部分Br指令是需要连续跳转的，因此BTB和BHT的正确预测率均很高，而Lab3CPU的预测正确率很低。由实验结果可以看出，矩阵乘法与BHT结合性能最佳。
6. 在大多数情况下，BHT的性能最佳