中国科学技术大学计算机学院
《计算机体系结构》实验报告

2021.06.10

实验题目：分支预测

学生姓名：胡毅翔

学生学号：PB18000290

计算机实验教学中心制

2019 年 9 月

1 实验目的

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

2 实验环境

1. PC 一台
2. Windows 10 操作系统
3. Vivado 2019.1
4. Visual Studio Code 1.56.2

3 BTB 实现

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。

在我们之前实现的 lab2 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。

主要设计参照如下图所示，其中 Branch PC 标记对应程序中的 BranchTagAddress 变量，Predicted PC 对应存储的预测结果对应程序中的 BranchTargetAddress 变量。当 IF 阶段产生的指令在 BTB 表中找到对应的一项并且最近一次跳转的标志位为真，则将预测结果有效变量改为 1，将预测的 PC 值从 BTB 表中取出。当 EX 阶段的操作码表示当前指令是 BR 类指令时。当 EX 阶段的 PC 值在 BTB 表中有存储时，更新 PC 值对应的预测 PC 值为 br_target 值，并将是否跳转的标示量更改为 br 对应的值；如果不存在对应的存储，则按照 FIFO 顺序进行 BTB 表的写入。

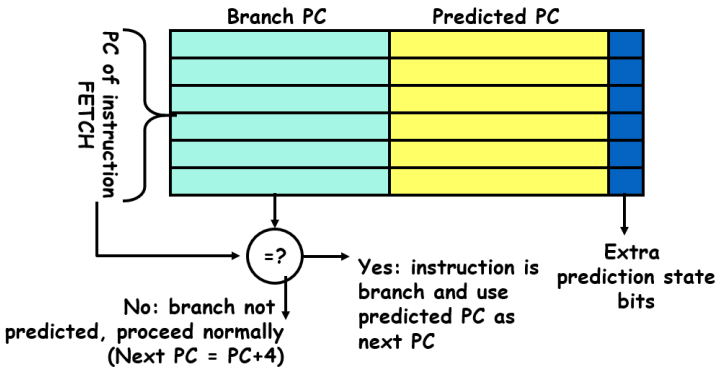

图 1: BTB

状态机如下图。

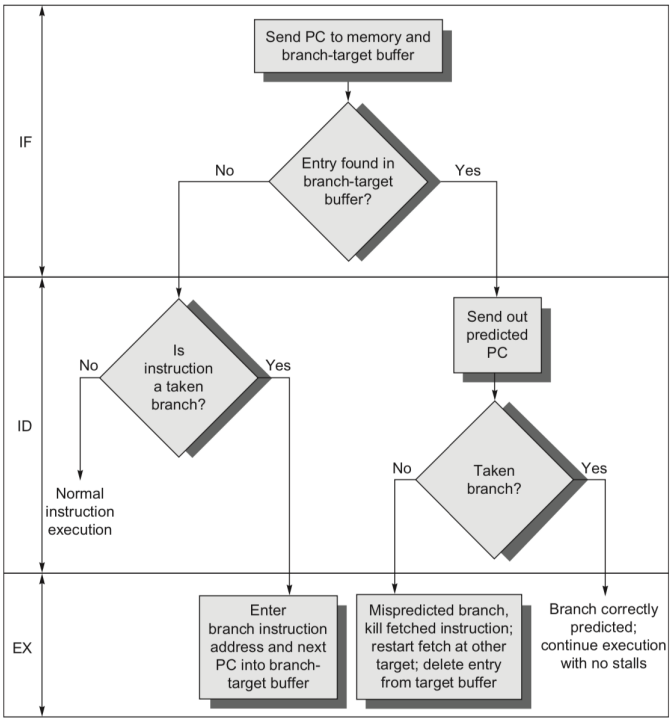

图 2: BTB 状态机

实现的代码如下:

```
module btb # (
    parameter ENTRY_NUM = 64//BTB大小
)(
    input clk,
    input rst,
    input [31:0] PC_IF,
```

```
    input [31:0] PC_EX,
    input [31:0] br_target,
    input br,
    input [6:0] Opcode_EX,
    output reg [31:0] PredictPC, //PC预测结果
    output reg PredictPCValid       //预测结果有效
);
//分支Opcode
localparam BR_OP = 7'b110_0011;
//Buffer define
reg [31:0] BranchTagAddress[ENTRY_NUM - 1 : 0];
reg [31:0] BranchTargetAddress[ENTRY_NUM - 1 : 0];
reg Valid[ENTRY_NUM - 1 : 0];
//FIFO Index
reg [15 : 0] Index;
//Produce PredictPC
always @ (*) begin
    if (rst) begin
        PredictPC <= 32'b0;
        PredictPCValid <= 1'b0;
    end
    else begin
        PredictPC <= 32'b0;
        PredictPCValid <= 1'b0;
        for (integer i = 0; i < ENTRY_NUM ; i++ ) begin
            if ((PC_IF == BranchTagAddress[i]) && Valid[i]) begin
                PredictPCValid <= 1'b1;
                PredictPC <= BranchTargetAddress[i];
            end
        end
    end
end
//Renew Buffer
always @ (posedge clk or posedge rst) begin
    if (rst) begin
```

```
        for (integer i = 0; i < ENTRY_NUM ; i++ ) begin
            Valid[i] <= 1'b0;
            BranchTagAddress[i] <= 32'd0;
            BranchTargetAddress[i] <= 32'd0;
        end
        Index <= 16'd0;
    end
    else begin
        if (Opcode_EX == BR_OP) begin
            integer i;
            for (i = 0; i < ENTRY_NUM; i++) begin
                if (PC_EX == BranchTagAddress[i]) begin
                    BranchTargetAddress[i] <= br_target;
                    Valid[i] <= br;
                    break;
                end
            end
            if (i == ENTRY_NUM) begin
                BranchTargetAddress[Index] <= br_target;
                Valid[Index] <= br;
                BranchTagAddress[Index] <= PC_EX;
                Index = Index + 1;
            end
        end
    end
end
endmodule
```

4 BHT 实现

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。如下图：

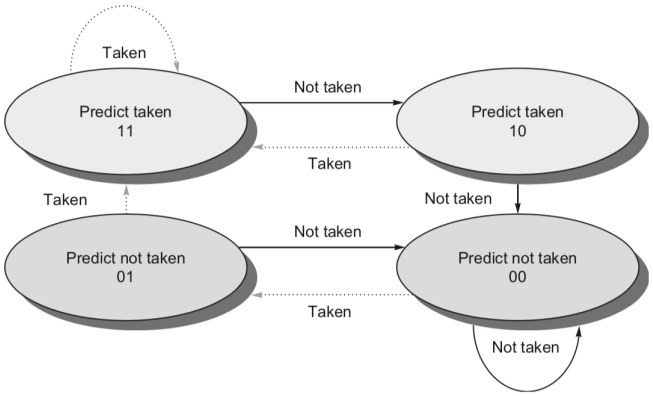

图 3: BHT 状态机

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。

实现的代码如下：

```
module bht (
    input clk,
    input rst,
    input [7:0] tag,
    input [7:0] tagE,
    input br,
    input [6:0] Opcode_EX,
    output PredictF //预测结果标志
);
//Branch Opcode
localparam BR_OP = 7'b110_0011;

reg [1:0] Valid[255 : 0] ;

assign PredictF = Valid[tag][1];

localparam SN = 2'b00;
```

```
localparam WN = 2'b01;
localparam WT = 2'b10;
localparam ST = 2'b11;

always @(posedge clk or posedge rst) begin
    if (rst) begin
        for (integer i = 0; i < 256; i++ ) begin
            Valid[i] <= WN;
        end
    end
    else begin
        if (Opcode_EX == BR_OP) begin
            if (br) begin
                Valid[tagE] <= (Valid[tagE] == ST) ? ST : Valid[tagE] + 2'
                    b01;
            end
            else begin
                Valid[tagE] <= (Valid[tagE] == SN) ? SN : Valid[tagE] - 2'
                    b01;
            end
        end
    end
end

endmodule
```

5 实验分析

5.1 实验统计

5.2 结果分析

1. 分支收益和分支代价与 CPU 设计有关，而与具体样例无关。
2. 未使用分支预测的周期数 = 指令数 + 错误预测数 × 预测错误惩罚。
3. 使用分支预测的周期数 = 指令数 + 错误预测数 × 预测错误惩罚。

样例	btb.S	bht.S	QuickSort.S	MatMul.S
分支收益（Cycle）	2	2	2	2
分支代价（Cycle）	2	2	2	2
未使用分支预测（Cycle）	508	533	68346	354610
使用 BTB 分支预测（Cycle）	312	383	69196	346816
差值（Cycle）	196	150	-850	7794
使用 BTB&BHT 分支预测（Cycle）		361	67782	346344
差值（Cycle）		172	564	8266
分支指令数	101	110	16178	4624
BTB 动态预测正确次数	99	86	14350	4076
BTB 动态预测错误次数	2	24	1828	548
BTB&BHT 动态预测正确次数		97	15057	4312
BTB&BHT 动态预测错误次数		13	1121	312

表 1: 实验统计结果

4. 差值 = 错误预测数差值 × 预测错误惩罚。

5. BTB 预测错误次数 = 循环个数 × 2。

6. BTB&BHT 预测错误次数（自 01 状态启动） = 循环个数 + 相同循环种数。

7. 使用 BTB&BHT 效果最佳，预测错误次数少，运行总周期数少。在一些情况下，预测错误次数过多，会导致分支预测的总周期数多于未预测的情况。

6 总结

通过本次实验，进一步加深了对课堂所学分支预测部分知识的理解，提高了 Verilog 的编程实践能力。