COL 216 ASSIGNMENT 2 REPORT
ARYAN GAURAV(2020CS10327)
ANKIT KUMAR (2020CS10323)

1.MIPS 5 STAGE WITHOUT BYPASS IMPLEMENTATION:

1.1 Data structure used:

For implementing this we had used the skeleton given for assignment 2 and added the functions in the same code which are described below:
As we know in the mips we have 5 stages that are IF ,ID,EX,MEM,WB so for each stage first declared a struct variable in which we used a vector that contains the information of the command which is to be executed by the corresponding state and all the structs are by default initialised by their default values set to null and also for the MEM stage their are some additional information we used such as the the address value for reading from memory,what value we have to write to latch 5 which then will be write back to register. The pictures of structures are given below:

```
struct ID_stage
{
vector<string>command;
vector<string>data;
};

struct ALU_stage
{
vector<string>command;
int value;
};

struct MEM_Stage
{
vector<string>command;
int adressvalue_dvalue;
int writedata;
};
struct WB_stage
{
   vector<string>command;
   int writedata;
};
```

Now for maintaining the clock we used int variable which is keep increasing in a while loop that is implemented to simulate the instructions and also used a int variable program counter for pointing to specific instruction to be complete and lastly used a storecomand vector which is used to terminate the program when all its instructions are completed.The snippets are given below:

```
vector<vector<string>>stored commands;
       PCcurr=0;
```

Now the control path and data path is done by the function named as five_stage_pipelined which is described below:

1.2 DATA PATH AND CONTROL PATH:

For implementing this we used a infinite while loop which executes each of the five stage from top to bottom i.e on the top is WB then MEM and so on in each iteration and also increase the clock count now implementation of each stage is described below:

(i)IF STAGE:

This stage just fetch the next command by incrementing the oc counter and then transfer that command to the ID stage when their is no stall ,no branching or no jumping status is true its snippet is :

```
   //**********************IF
STAGE************************************************************
    if(PCcurr>=commands.size())  //encountered all commands;
    {
       if(storedcommands.empty()==true)
       {
           printRegistersAndMemoryDelta(clockcycle);
           break; //program completed
       }

       continue;


     }

    vector<string>command=commands[PCcurr];



    if(stall_count==0 && branch_stall==false)
       {  storedcommands.push_back(command);
           latch2.command=command;
```

```
            if(command[0]=="beq" ||command[0]=="bne" || command[0]=="j")
               branch_stall=true;
             PCcurr++;
       }
```

The first condition here in the snippet is for checking the termination of the program and the second snippet is used for transferring the command to the ID in the next cycle.

(ii)ID STAGE :

This stage just decodes the command it receives from if stage and after checking any dependency of the instructions it just pass the control to the ALU stage its snippet is:

```
//*****************************DECODE
STAGE************************************************
   if(!latch2.command.empty())
   {
     if(latch2.command[0]=="j")
     {
       instructions[latch2.command[0]](*this, latch2.command[1],
latch2.command[2], latch2.command[3]);
       branch_stall=false;
       latch2.command.clear();
       storedcommands.pop_back();
       continue;
     }


     if(stall_count!=0)
             stall_count--;


     if(stall_initiated==false)
     {
       if(latch2.command[0]=="add" || latch2.command[0]=="sub" ||
latch2.command[0]=="mul" || latch2.command[0]=="slt" ||
latch2.command[0]=="addi" || latch2.command[0]=="lw" ||
latch2.command[0]=="sw" || latch2.command[0]=="beq" ||
latch2.command[0]=="bne")
       {
           string consumer1=latch2.command[2];
           string consumer2=latch2.command[3];
```

```
          if(latch2.command[0]=="lw" || latch2.command[0]=="sw")
           {
              if(latch2.command[2].back() == ')')
               {
                  int lparen = latch2.command[2].find('('), offset =
stoi(lparen == 0 ? "0" : latch2.command[2].substr(0, lparen));
                  consumer2 = latch2.command[2].substr(lparen + 1);
                  consumer2.pop_back();
               }

              else
              consumer2="M";

           }

           if(latch2.command[0]=="sw")
            {
              consumer1=latch2.command[1];
            }

           if(latch2.command[0]=="lw")
            {
              consumer1="M";
            }

           if(latch2.command[0]=="beq")
            {
              consumer2=latch2.command[1];
            }

           if(latch2.command[0]=="bne")
            {
              consumer2=latch2.command[1];
            }



           string producer;
           stall_initiated=true;
```

```
        if(!latch4.command.empty())
        {
           if(latch4.command[0]=="add" || latch4.command[0]=="sub" ||
latch4.command[0]=="mul" || latch4.command[0]=="slt" ||
latch4.command[0]=="addi" || latch4.command[0]=="lw")
               producer=latch4.command[1];

           if(producer==consumer1 || producer==consumer2)
              {
               stall_count=stall_count+2;
               continue;

              }

        }

        if(!latch5.command.empty())
        {
           if(latch5.command[0]=="add" || latch5.command[0]=="sub" ||
latch5.command[0]=="mul" || latch5.command[0]=="slt" ||
latch5.command[0]=="addi" || latch5.command[0]=="lw")
               producer=latch5.command[1];


           if(producer==consumer1 || producer==consumer2)
              {

               stall_count=stall_count+1;
               continue;

              }

        }

       }
    }

       //transfering the command value from it
          if(stall_count==0)
```

```
            {
                latch3.command=latch2.command;
                stall_initiated=false;
                latch2.command.clear();
            }

    }
```

Here in this there are three section first one is that if a jump instruction is their then solve its value in this stage only and the second middle section is for stall checking as in mips there are three cases of stalls between two consecutive instructions and between alternate instructions so these all are handled in the middle section. Then in the final third section after checking everything passing the control to the ALU stage whose snippet is given below:

Here for checking the stalls we just checked the further two stages of it in which we find for the point of production which might be the consumer point for the current instruction which will result in stall. So if the point of production is found on just the next stage then the stall must be of 2 and if the point of production that is dependent on the point of consumer of the current instruction is just next to next stage then the stall count must be 1 and for all other cases the stall count is zero as the data will always be available for the alu to be read correctly.

(iii)ALU STAGE:

```
//*****************************ALU
STAGE************************************************
    if(!latch3.command.empty())
    {
        latch4.command=latch3.command;
        if(latch3.command[0]=="sw")
        {
            int
adr=sw(latch3.command[1],latch3.command[2],latch3.command[3]);
            latch4.adressvalue_dvalue=adr;
            int writedata=registers[registerMap[latch3.command[1]]];
```

```
            latch4.writedata=writedata;
        }

        if(latch3.command[0]=="lw")
        {
            int
adr=lw(latch3.command[1],latch3.command[2],latch3.command[3]);
            latch4.adressvalue_dvalue=adr;
        }

        if(latch3.command[0]=="add" || latch3.command[0]=="sub" ||
latch3.command[0]=="mul" || latch3.command[0]=="slt" ||
latch3.command[0]=="addi")
        {
            int result=instructions[latch3.command[0]](*this,
latch3.command[1], latch3.command[2], latch3.command[3]);
            latch4.adressvalue_dvalue=result;

        }


        if(latch3.command[0]=="beq" || latch3.command[0]=="bne")
        {
            instructions[latch3.command[0]](*this, latch3.command[1],
latch3.command[2], latch3.command[3]);
            branch_stall=false;
            latch3.command.clear();
            latch3.value=0;
             continue;

        }

        latch3.command.clear();
        latch3.value=0;
    }
```

In this stage after receiving the control from the decode stage doing operation according the opcode of each instruction and then passing control to memory stage whose snippet is given below:

(iv)MEM STAGE:

This stage does 3 kind of thing one is if the instruction was of type load then it just takes the address calculated from the ALU stage takes the data out and pass it to the stage 5, if the instruction is type of store word then it just stores the data to the memory at the address computed by the previous stage and if the instruction is type of add , addi mul then it just pass the value come from previous stage to the next stage.

```
//*************************************MEM_STAGE************************
*************************
   if(!latch4.command.empty())
   {
      latch5.command=latch4.command;

      if(latch4.command[0]=="add" || latch4.command[0]=="sub" ||
latch4.command[0]=="mul" || latch4.command[0]=="slt" ||
latch4.command[0]=="addi")
       {
        latch5.writedata=latch4.adressvalue_dvalue;
       }

       if(latch4.command[0]=="sw")
       {
         data[latch4.adressvalue_dvalue]=latch4.writedata;
         memoryDelta[latch4.adressvalue_dvalue]=latch4.writedata;
       }

       if(latch4.command[0]=="lw")
       {
         latch5.writedata=data[latch4.adressvalue_dvalue];
       }

     latch4.command.clear();
     latch4.writedata=0;
     latch4.adressvalue_dvalue=0;
   }
```

(V) WB STAGE :

This stage just writes the data that from the previous stage to the corresponding register and this is the last step of our one instruction so after this it will be deleted from the stored commands vector which we used in the simulation its snippet is:

```
//*****************************************WB_STAGE*********************
*****************************
   if(!latch5.command.empty())
    {
       if(latch5.command[0]=="add" || latch5.command[0]=="sub" ||
latch5.command[0]=="mul" || latch5.command[0]=="slt" ||
latch5.command[0]=="addi" || latch5.command[0]=="lw")
       {
           registers[registerMap[latch5.command[1]]]=latch5.writedata;
       }

       latch5.command.clear();
       latch5.writedata=0;
       storedcommands.erase(storedcommands.begin());

    }
```

2.MIPS 5 STAGE WITH_BYPASS IMPLEMENTATION:

For this implementation everything used is same as that of the mips 5 stage without bypass but just used slight modification in the struct of the ALU stage as to handle the forwarding and also added the change in implementation of ID stage and ALU stage.

(i)For id stage: to handle the forwarding we first check the dependency of the current instruction by checking the point of production in the next stage and if we found out that the point of production is same as that of point of consumption then i marked the corresponding registers whose value will be forwarded in the alu stage as true so in the alu instead of taking the values from the registers it will take the value from marked forwarded stage and in this way we are able to reduce the stall count.Their will be stall of 1 unit when the instruction type of like this occur:
lw $1,8($2)
addi $2,$3,1000 this is also handled in the bypass code.

(ii)For alu stage : we had modified its behaviors such that whenever the value is forwarded which is checked by the bool variables defined in the struct of the alu stage we will take the value from corresponding latches and otherwise it will do its function as usual.

```
struct forwarding{
   int reg_1_value;
   int reg_2_value;

   bool reg1_forwarding;
   bool reg2_forwarding;
};
```

```
struct ALU_stage
{
vector<string>command;
int  value;
struct forwarding Mydata;

};
```

Modified struct variables:

```
   //*****************************DECODE
STAGE************************************************
   if(!latch2.command.empty())
   {
     if(latch2.command[0]=="j")
     {
       instructions[latch2.command[0]](*this, latch2.command[1],
latch2.command[2], latch2.command[3]);
       branch_stall=false;
       latch2.command.clear();
       storedcommands.pop_back();
       continue;
     }
```

```
    if(stall_count!=0)
            stall_count-;

       if(!latch4.command.empty())
       {
          if(latch4.command[0]=="add" || latch4.command[0]=="sub" ||
latch4.command[0]=="mul" || latch4.command[0]=="slt" ||
latch4.command[0]=="addi" || latch4.command[0]=="lw" )
              producer=latch4.command[1];

          if(producer==consumer1)
             {
              latch3.Mydata.reg1_forwarding =true;
              latch3.Mydata.reg_1_value = latch4.adressvalue_dvalue;

              if(latch4.command[0]=="lw" && latch2.command[0]!="sw")
                 {
                  // latch3.Mydata.reg_1_value = latch5.writedata;
                  stall_register=1;
                   stall_count=1;
                   stall_initiated=true;
                   continue;
                 }
```

Modified decoder function:

Note: Here the decoder function is only shown for the stall checking as the full code is large so i had just attached the snippet of it in which i am checking the consumer value in producer

```
   //*******************************ALU
STAGE*************************************************
   if(!latch3.command.empty())
 {
```

```
        // cout<<"clock :: "<<clockcycle<<latch3.command[0]<<endl;
        latch4.command=latch3.command;

        if(latch3.command[0]=="sw")
            {

                sw(latch3.command[1],latch3.command[2],latch3.command[3]);

                if(latch3.Mydata.reg1_forwarding){
                    latch4.writedata = latch3.Mydata.reg_1_value;
                    latch3.Mydata.reg1_forwarding= false;
                }
                else{
                    latch4.writedata =
registers[registerMap[latch3.command[1]]];
                }
            }

        if(latch3.command[0]=="lw")
        {
            if(latch3.Mydata.reg2_forwarding){
                My_Address(latch3.command[2]);
                latch3.Mydata.reg2_forwarding = false;
            }
            else{
            int
adr=lw(latch3.command[1],latch3.command[2],latch3.command[3]);
            latch4.adressvalue_dvalue=adr;
            }
        }
```

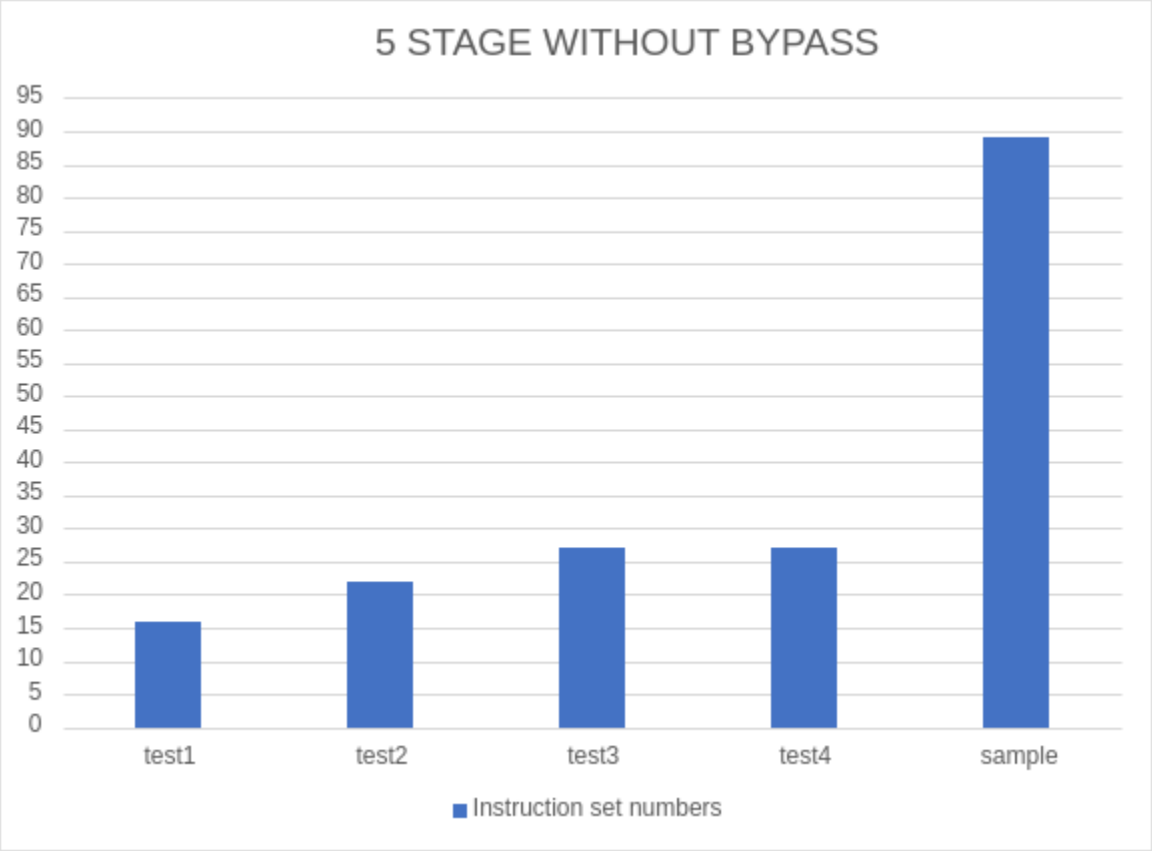
Modified alu function

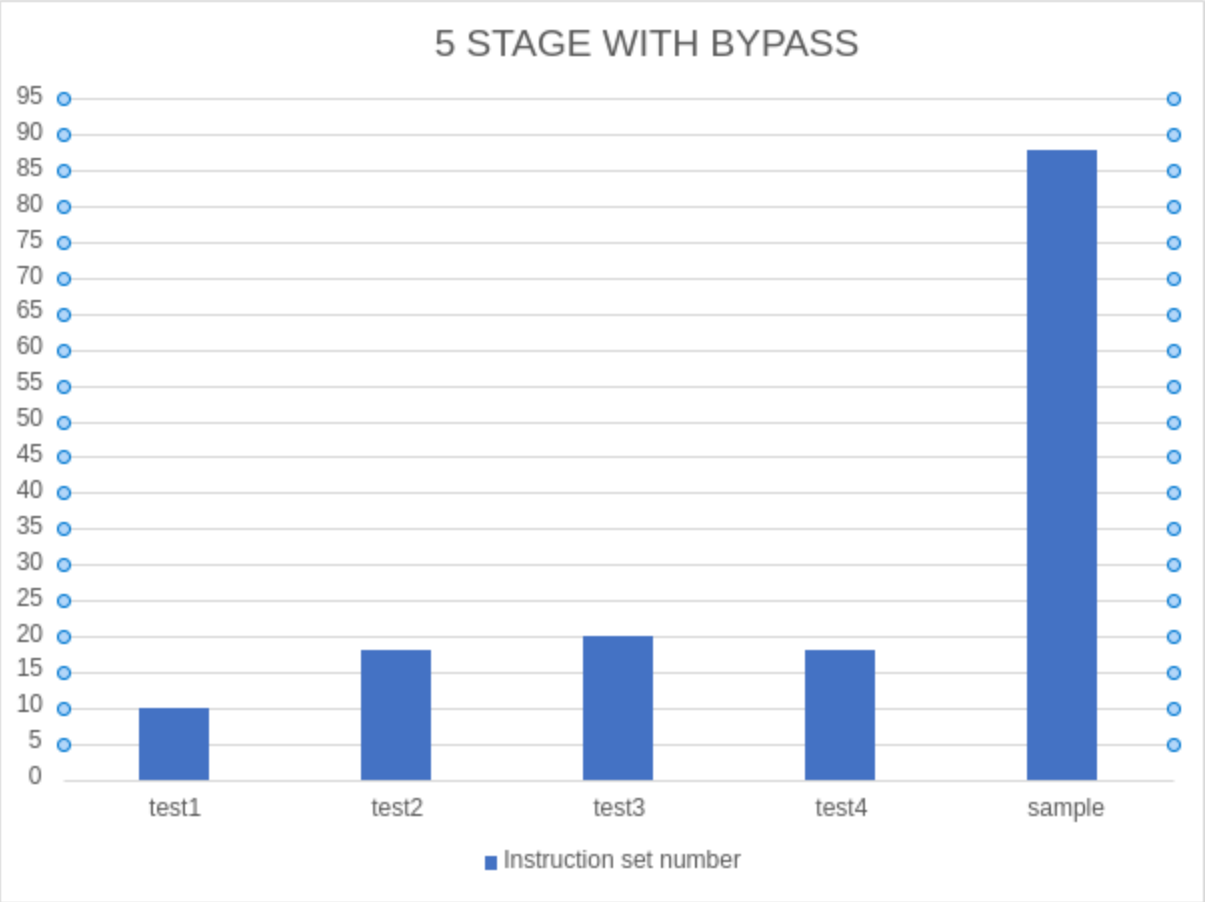
Note: Here the alu function is only shown for a beq operation but all the other operations are also like this happen in which if a value is forwarded then it will use the forwarded value otherwise it will use the register value.

For all the other stages the procedure is same as they are not affected by any dependencies so no possibility of stalls in them.

4. PERFORMANCE CHARTS FOR EACH IMPLEMENTATION:

4.1 5stage without forwarding:





Explanation: For the above plots we can clearly see that the cycle numbers with bypassing is always less than or equal to the without bypassing. In test 1 there is a drop of around 8 cycles and in test 2 drop is only of 2 cycles in test 3 also dropped by 2 and in the last sample the drop is only one. So this is because in forwarding also there will stalls occurs if in our program multiple loads instruction are there followed by any instruction which stores the final data to the registers. Because in these cases the forwarding is only possible after 1 stall. Thats why in the 4th test cases the difference is not large enough as the load instructions are many, and in test 1 as these type of instructions are less the difference is huge.

7-9 stage we were unable to implement*.

3.Branch Predictor:

We have implemented the three classes SaturatingBranchPredictor, BHRBranchPredictor and SaturatingBHRBranchPredictor of the file BranchPredictor.hpp as stated in the assignment. The table of output-percentage-accuracy is as follows:

Prediction Strategy:	value (start state):	size (combination array):	Percentage Accuracy (%):
Saturating Counters	0	N/A	79.0146%
Saturating Counters	1	N/A	83.9416%
Saturating Counters	2	N/A	87.9562%
Saturating Counters	3	N/A	86.6788%
BHR	0	N/A	71.5328%
BHR	1	N/A	72.2628%
BHR	2	N/A	72.6277%
BHR	3	N/A	72.8102%
Saturating Counters+BHR	0	pow(2,16)	79.562%
Saturating Counters+BHR	1	pow(2,16)	81.0219%
Saturating Counters+BHR	2	pow(2,16)	87.5912%
Saturating Counters+BHR	3	pow(2,16)	87.2263%
Saturating Counters+BHR	0	pow(2,10)	80.292%
Saturating Counters+BHR	1	pow(2,10)	82.6642%
Saturating Counters+BHR	2	pow(2,10)	87.5912%
Saturating Counters+BHR	3	pow(2,10)	86.8613%
Saturating Counters+BHR	0	pow(2,7)	77.1898%
Saturating Counters+BHR	1	pow(2,7)	79.927%
Saturating Counters+BHR	2	pow(2,7)	84.6715%
Saturating Counters+BHR	3	pow(2,7)	85.219%

Figure 1: Percentage Accuracy Table

Brief overview of the class implementations are as follows:

3.1 SaturatingBranchPredictor:

The table vector is indexed on the last 14 LSBs of the pc and the value at each index is a saturating counter corresponding to that pc, which ranges from 00 to 11. Now, we have two functions to implement: bool predict(uint32 t pc) and void update(uint32 t pc, bool taken). In the predict function, we extract the last 14 LSB bits value from the pc and use it as index to find the corresponding saturating counter in the table; once we find it, we check if it's value is < 2, if yes, then the branch pred variable is set to f alse else true. branch pred is our return value. 1 Next we have the update function. Here, we update the value of the saturating counter corresponding to the given pc. We first extract the index from pc and then, check whether the actual branch decision, i.e., taken was true or f alse. In case it is true, we check the value of the saturating counter corresponding to the above extracted index; if it is < 3, we increment it by 1, else (=3) , we let it be 3. Similarly, when taken is false, we check the saturating counter's value, if it is > 0, we decrement it by 1, else (=0), we let it be 0.

3.2 BHRBranchPredictor:

Here also, we have a bhrT able vector of size 4 since the bhr bitset is of 2-bit size. Here the indexing of the bhrTable is done on the bhr value. For the predict function, we go the entry of the table corresponding to our current bhr value and check if it's < 2; if yes, we set branch pred to f alse, else true. The entries of the bhrTable are themselves saturating counters ranging from 00 to 11. In the update function, we check if the taken variable is true; if yes, we then check the value corresponding to the bhr value (as index), in the bhrTable. If it is < 3, we increment it by 1

else (=3), we let it be 3. Similarly, when taken is false, we check the bhrTable's entry corresponding to the current bhr value; if it is > 0, we decrement it by 1 else (=0), we let it be 0. It is only after updating the bhrTable, that we update the bhr value to store the updated branch history, i.e., the branch that happened now and the previous branch. To do so, we left shift the bhr bitset by 1 position and set the LSB to taken. The accuracy of BHR comes to be less than that of saturating − counters because here, we are only taking the last 2 branches into consideration while saturating counters consider the branch history of each branch instruction.

3.3 SaturatingBHRBranchPredictor:

Here we have the bhrT able, bhr value, saturating counters table and a combination vector whose size has to be <= pow(2, 16). For our combination algorithm, we update the bhr as well as saturating counter parameters (i.e., bhrTable, bhr value and table) as we did in the upper two versions and then we have our combination vector. It is indexed on the following: a 16-bit value formed by concatenating bhr value with last 14 LSBs of pc. We further take the moduli of this value with combination.size() (because a total of pow(2,16) permutations possible for the 16-bit value but the max value of size of combination vector is pow(2,16), size can be less than the max too). As we get the index from the above procedure, we go to the corresponding entry in the combination vector and update it's value. Now, updation of combination vector's value is based on the table shown in F igure − 2. This updation is based on giving saturating counters a higher priority than bhr (as we saw lower percentage accuracies in case of bhr). Since both saturating counters and bhr can have 4 states: strongly taken (st 11), mildly taken (mt 10), mildly not taken (mnt 01) and strongly not taken (snt 00), a total of 16 combinations of the above are possible. But we need 2 to form groups of 4 and assign one-one value to each group since the counter of combination vector itself is a 2-bit bitset. Note: Examples of ”giving higher priority to saturating counters”, say, saturating counter's state is mt(10) and bhr's state is mnt(01), then the combined state would be mt(10). Similarly say, sat counter's state is st(11) and bhr's state is snt(00), then output state would be mt(10) etc. (Rest given in Figure-2).

Giving SAT_COUNTER higher priority over BHR

BHR::	SAT_COUNTER::	SUM::	COMBINATION_COUNTER::
00	00	0	00
01	00	1	00
10	00	2	00
11	00	3	00
00	01	1	01
00	10	2	01
01	01	2	01
10	01	3	01
00	11	3	10
01	10	3	10
01	11	4	10
11	01	4	10
10	10	4	11
10	11	5	11
11	10	5	11
11	11	6	11

Figure 2: Combination Vector Updation policy

For the predict function, we find the index for the combination vector as specified above and check the counter's value: if < 2, branch pred is set false, else true. One thing to note is that the combination vector's entry corresponding to the index that we found above, is updated on the new/updated values of bhrTable and saturating counter's table, BUT the index of bhrTable, table and combination vector are found on the current values of bhr and pc (only after this is the bhr updated).

3.3 Summary and Advanced ideas:

We see that ONLY-saturating counters and Combination of BHRs with saturating counters give higher percentage accuracy as compared to BHRs-ONLY. We stated the reason for difference between ONLY-sat counters vs ONLYBHRs. For the higher percentage of Combination as compared to BHRs-only, it is intuitive because more history/information is taken into consideration in combination as compared to the latter. But, the percentage accuracies of only saturating counters and combination are nearly the same for the branchtrace.txt 3 provided. But in some start states, our combination algorithm does exceed the percentage accuracy of "ONLY-saturating counters". Our combination algorithm does provide an optimization on size of the combination vector, i.e., taking moduli does help in utilizing the vector, but only when our bhr.pc % size values are sparse/ distributed across the vector. If they get concentrated in a small region of the vector (i.e., modulus (s) results in same values again n again), we would have un-utilized space and high prediction inaccuracies because numerous branches would try

to predict on the same entry of the comb. table which would be logically wrong. A cure for the above could be that we introduce a visited vector, whose each index corresponds to the corresponding entry of the combination vector and each entry of the visited vector stores a pair of < visited bit, pc >. Now, this new vector is intialized with all zeroes and whenever a bhr.pc gets mapped to an entry in the combination vector, we change the visited bit to 1 and pc to the last 14 LSBs of the pc of the branch instruction. Next, whenever a branch instruction gets mapped to the same entry, we check the visited bit; since it is 1, we check if pcs' are the same, if yes, then that entry is the one we are looking for; if no, then we traverse the combination vector from that index to the next indices (if we reach the end, we go back to index 0 and keep going) till we either find an entry with visited bit 1 and pc equal to the pc of the current branch instruction OR visited bit 0 ; in the former case , this is the entry we are looking for whereas in the latter case, we store our new branch entry here, set visitied bit to 1 and the algorithm continues. In case non of the above happens, it means our combination table is full and so, as an alternative, we can go the saturating counter's table (with pow(2,14) entries) and predict and update according to that entry itself. Since we weren't allowed to introduce any new struct variables, we didn't implement the the visited vector.

4.WORK SPLIT:

For the 5 stage both implementations we both sat together to think upon the details of the implementation and the implementation was done by Ankit kumar and the branch predictor too we both together about the ideas of the implementation and code was done by Aryan Gaurav and report was made by both.