The American University in Cairo

Tomasulo Simulator

Project 2 – Computer Architecture

report

Aya Shaker 900160580

Habiba Gamal 900151007

Ali El-Said 900150264

Ahmed Wael Fahmy 900160127

Contents

1. Introduction:

This project simulates a dual-issue processor that implements Tomasulo's algorithm. Our implementation is in C++.
The processor also supports dual write and dual commit. It has several functional units: 2 load, 2 store, 3 JMP/JALR/RET, 2 BEQ, 3 ADD/SUB/ADDI, 1 NAND and 2 MUL functional units. We assumed a one-to-one mapping between each reservation station and a functional unit.
By this, our simulation does not support except certain instructions. These are lw, sw, beq, jalr, jmp, ret, add, sub, addi, nand and mul.
The processor is divided into 5 stage: fetch, issue, execute, write and commit.
Our implementation has 3 branching modes. 0 is static branch prediction, 1 is dynamic 1 bit branch prediction and 2 is dynamic 2 bit branch prediction.

2. Description of the Design:

a. Input:

Input takes place through 3 ways.

1. From a file: the instructions should precede the data, Instructions start by ".text:", then newline then "@<address>", then newline then the instructions in the same format as the project handout.
   The data section starts by ".data:" then a newline then "@<address> data" for each data item, which are separated by a newline.
2. From input stream. The command line prompts you on what you need to do. You are able to specify the starting address of the instructions and each address of each data item.
3. From a list. The command line prompts you on what you need to do. You are able to specify the starting address of the instructions and each address of each data item.

b. Time Tracing Table:

In order for us to be able to be able to know when we fetched, issued,executed, written back and committed the instruction, a struct was created that holds the clock time of when each stage happened:

```
struct ClockTracing
{
   int count;
   int fetch;
   int Issued;
   int Executed;
   int Written;
   int commit;
   parsedInst par;

};
```

This struct also has a counter which is set to the number of execution cycle it needs in order to be executed , which is then subtracted from, so it is used as a way to keep track of when the execution is done. parsedInst par holds the instruction associated with the rest of the table. In conclusion this table was created as a way to track that we are on the right track and that everything works accordingly.

c. Labels bonus:

We support branches and jumps to a label. The labels are detected when reading files by detecting any string that ends with colon, and this label is not ".text:" or ".data". When a a label is detected, it is stored in a map along with the pc (address of the following instruction).

During parsing, we check the immediate field of the BEQ and JMP instructions in the instruction string. If the immediate field is not a number, we look for the label in the map and insert in the immediate field in the struct of the parsed instruction as the address attributed with the label in the map.

d. Branch prediction bonus:

3 branch prediction techniques were implemented.The prediction module takes the number of prediction bits as an input (0,1,2) with 0 being static prediction.

- For static branch prediction, negative immediate generates a taken prediction and positive immediate generates a not-taken prediction. The prediction is stored in a vector to be able to compute the miss-prediction rate. When the branch instruction is committed, the true branch outcome is compared with the branch prediction that is stored in the vector to be able to compute the misprediction rate.

- For one bit prediction, the function also has to modes that are indicated by a bool. Prediction model predicts taken the first time, and subsequent times the prediction is the same as the last result. Update mode inputs the actual branch result to keep the vector up to date.

- Two bit prediction is similar except it has 4 states (Strongly Taken, Taken, Not Take, Strongly Not Taken). The initial prediction is Taken. Predict mode outputs either taken or not taken while update mode works like a Finite State machine, with each of the aforementioned states being updated depending on the actual branch result.

e. Fetch:

We have an instruction buffer that is implemented as a queue of size 4 in C++. If the instruction buffer is empty, it is loaded by 4 instructions from pc until pc + 3. This simulates the behavior of an instruction cache that has 4 words per block.
During this stage, we also fill the vector of the instructions for the time tracing table with the fetched instructions.
When the instruction is fetched, the time tracing table is updated for this particular instruction (checked by its address). The fetch time is the current clock time.

f. Issue:

In this stage, the front of the instruction queue is checked. If there is an empty entry in the ROB, the required reservation station is checked. If there is an empty entry in it, the instruction is issued. When the instruction is issued, its entry in the time tracing table is updated such that the issue time is the current clock time. It is checked that the issuing takes place after the fetching and not in the same clock cycle.

g. Execute:

After the instruction gets issued, it is then sent to the execution stage where then they are executed on based on the number of reservation stations that this instruction has. Load, store, multiplication and branching all have 3 reservation stations, therefore they were grouped together in the same for loop , where then we check the reservation stations for each instruction. The first if statement accesses the first location in the load's reservation station where it checks that Qj has a value of negative one and at the same time that that reservation station is busy in order to ensure that there is something inside the designated reservation station. Then we try to find the address of the instruction in the reservation station that corresponds to the address in the Trace table. After we find the instructions address in the table, we check first of all that the instruction has already been issued , and that the execution starts after issuing and not at the same clock cycle. The instruction has been issued in. Inside of this if statement is when we start implementing the execution, so at first it checks that the clock count in the table is -1, and if it is it goes inside the if statement and sent the clock count for the execution , and every clock cycle in comes back to the for loop and then it's supposed to enter the second if statement where it check if the count does not equal to 0 and if doesn't it subtracts one from the count , and finally after the clock count is done it enters the third if statement where it checks that count is 0 and that the execution clock is still -1 where then it sets the execution clock with the current clock value and executes the instruction, and all the other instructions follow the same procedure except they access different reservation stations, and the clock counts are set to different numbers.

h. Write:

In this stage, the time tracing table is checked, so that if there are instructions (from 0 to 2) are ready to be written. In other words, finished their execution, their writing will start. The writing cannot start in the same cycle as the execution.
If there are 3 instructions that are ready to be written, only the first 2 instructions are written.

In the write stage, the result of the instruction from the execution stage is written to the value column of the instruction’s entry in the ROB. For load, store, beq, jalr, jmp, ret, the target address that is computed in execution is written to an address column in the ROB in the entry of the instruction.

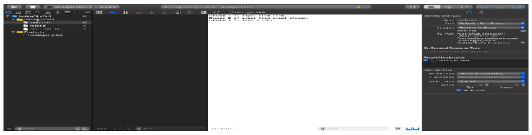
i. Commit:

In this final stage, we commit the instructions. As this is a dual issue processor, we must also be able to commit two instructions at a time. Commission is always done after writing by one cycle if there, as if there cannot be a dependency as we always commit the instruction(s) at the ROB head. Flushing is also handled in the commit stage, specifically for JALR, RET, and BEQ. This is also where we update the branch prediction table in case it is a BEQ instruction. After we commit we update the ROB head position and check if it is completely empty. This check, along with others, is used to terminate the program when it is done.

3. Assumptions:

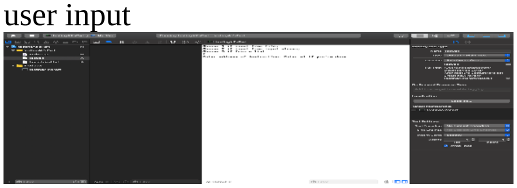
RAW dependent hazards do not resume execution in the same cycle as the independent instruction writes.
The structural hazards do not resume issuing in the same cycle when the functional unit was emptied.
The full ROB hazard does not resume issuing in the same cycle when an ROB entry was emptied.
The branch prediction is used in the fetch stage. It is used to fetch the target address of the prediction.
The actual branch outcome is acted upon in the commit stage.
JALR and RET are handled (pc jumps to their target) in the commit stage.
JMP is handled in the fetch stage (similar as branch prediction)
For JALR, RET and mispredicted branches, the processor flushes in their commit stage.
For a load dependent on a store, it does not start execution until the store commits.

4. Step-by-step simulation:

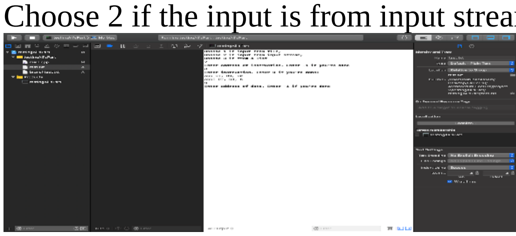


Step 1: Input type of input

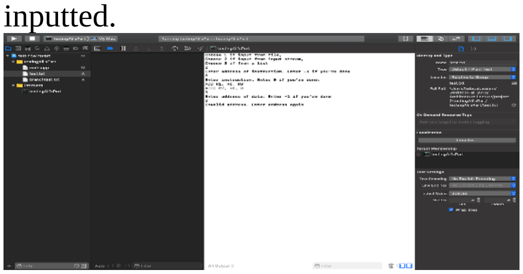
If input is from file, input the filename, the program continues without any other user input



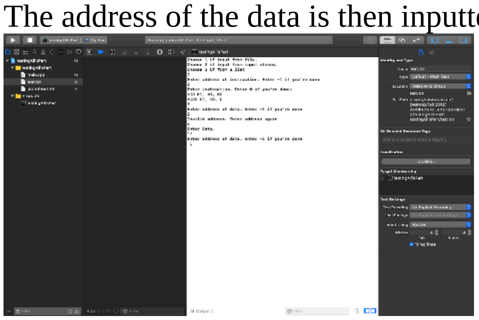
Choose 2 if the input is from input stream



As noticed, the command line prompts the user with all the necessary inputs.
In this figure, the address of the instruction is inputted, then the instructions are inputted.

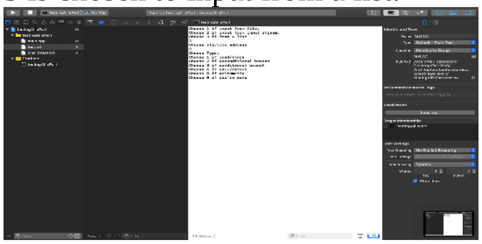


The address of the data is then inputted.

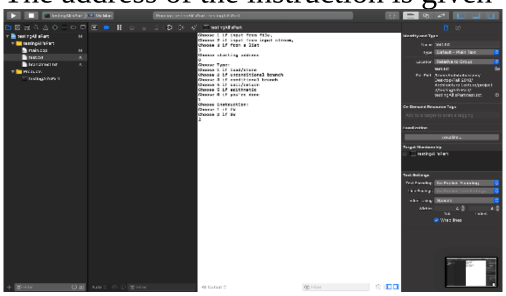


Data is inputted, then -1 is inserted to stop insertion of data.

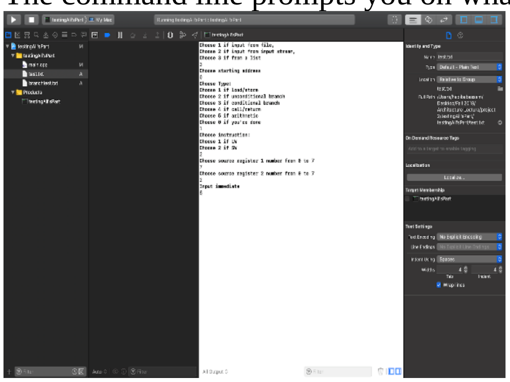
3 is chosen to input from a list.



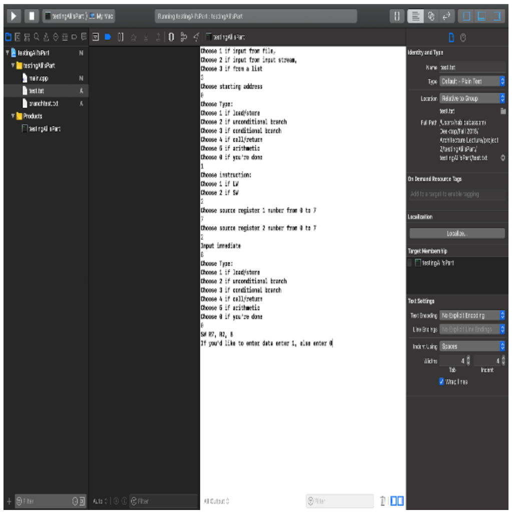
The address of the instruction is given



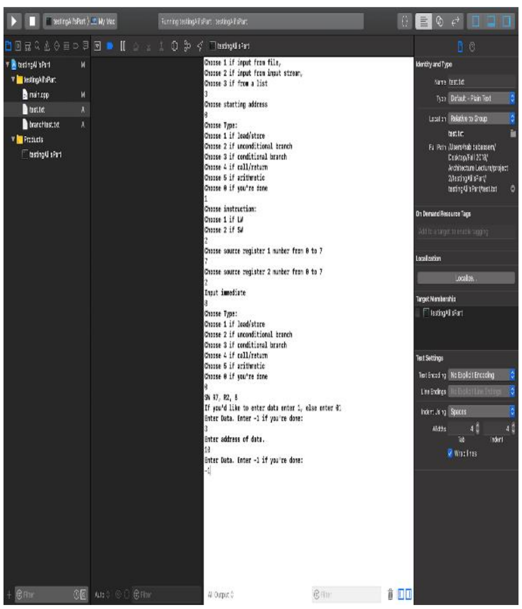
The command line prompts you on what you need to input to choose the instruction.



Registers and immediate are inputted.



The instructions that you selected are then inputted to be more visually clear to the user.

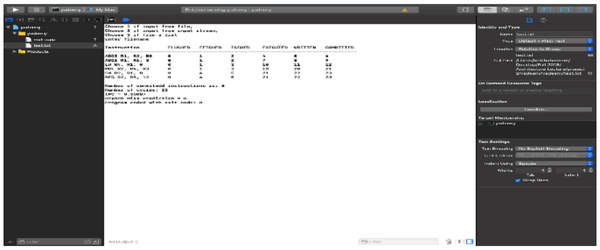


Then the command line prompts the user to input the data followed by its address.

5. Results of each test-case:

a. Test Case 1:

1. Result:



2. Discussion: Our program assumes that we read the instructions from a cache whose block contains 4 words. This is how we fetch 4 instructions at a time. The first 4 instructions are fetched at cycle 1. The first 2 instructions are issued at cycle 2 since the ROB is empty and reservation stations are empty, and the processor supports dual-issue.

Each instruction has a RAW hazard with its preceding instruction. Therefore, each instruction starts its execution after the write of the preceding function is completed. The last 2 instructions both execute in parallel as sw does not write to a register, so BEQ is not RAW dependent on it. The BEQ finishes before the store instruction, however they both commit in the same cycle due to the ROB.

b. Test Case 2:

1. Result:

Instruction	FLUSHED	FETCHED	ISSUED	EXECUTED	WRITTEN	COMMITTED
ADDI R1, R1, 0	0	1	2	4	5	6
ADDI R2, R2, 2	0	1	2	4	5	6
ADDI R3, R1, 300	0	1	3	7	8	9
LW R5, R3, 0	0	1	3	10	11	12
ADDI R6, R5, 1	0	4	6	13	14	15
ADDI R3, R3, 1	0	4	6	10	11	15
SW R6, R3, 0	0	4	7	15	16	17
ADDI R1, R1, 1	0	4	9	11	12	17
BEQ R2, R1, EXIT	0	10	11	13	14	18
JMP Loop	0	10	13	14	15	18
ADDI R3, R1, 300	0	10	16	18	19	20
LW R5, R3, 0	0	10	16	21	22	23
ADDI R6, R5, 1	0	17	18	24	25	26
ADDI R3, R3, 1	0	17	18	21	22	26
SW R6, R3, 0	0	17	19	26	27	28
ADDI R1, R1, 1	0	17	20	22	23	28
BEQ R2, R1, EXIT	0	21	22	24	25	29
JMP Loop	1	21	24	25	26	-1
ADDI R3, R1, 300	1	21	27	29	-1	-1
LW R5, R3, 0	1	21	27	-1	-1	-1
ADDI R6, R5, 1	1	28	29	-1	-1	-1
ADDI R3, R3, 1	1	28	29	-1	-1	-1
SW R6, R3, 0	1	28	-1	-1	-1	-1
ADDI R1, R1, 1	1	28	-1	-1	-1	-1

```
Number of commitedd instructions is: 17
Number of cycles: 29
IPC = 0.586207
Total Branch Count = 2
Branch flushed = 0
Branches = 2
Branch Miss Prediction = 1
Branch Miss Prediction Rate = 0.5
Program ended with exit code: 0
```

2. Discussion: This test case is an assembly code of the following C++ code:

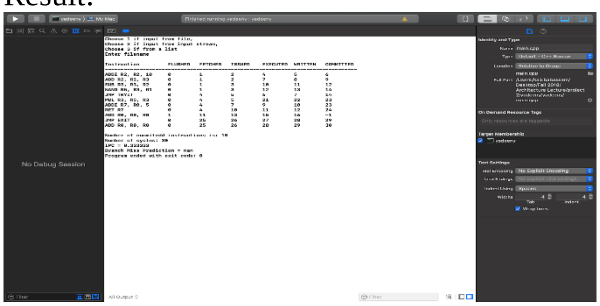
For(int i=0;i<4;i++)
{
A[i+1]=A[i]+B[i+1];
}
This test case shows a Loop that loads a value from the memory at address 300 then adds to it to array B which is located at address 400 and then stores it back in the next address in A , when the i is equal to 4 then the rest of the instructions after the Branching are all flushed, because Branch gets committed and we jump directly to Exit, therefore once branch is taken all the preceding instructions are flushed and none are committed.

c. Test Case 3:
   1. Result:

```
ADDI R1, R1, 0      0     1     2     4     5     6
ADDI R2, R2, 4      0     1     2     4     5     6
ADDI R3, R1, 300    0     1     3     7     8     9
ADDI R6, R3, 1      0     1     6     10    11    12
ADDI R5, R1, 400    0     7     8     10    11    12
LW R3, R3, 0        0     7     8     10    12    13
LW R5, R5, 0        0     7     9     13    14    15
ADD R7, R3, R5      0     7     9     16    17    18
SW R7, R6, 0        0     10    11    18    19    20
ADDI R1, R1, 1      0     10    13    15    16    20
BEQ R2, R1, EXIT    0     10    13    17    18    21
JMP Loop            0     10    14    15    16    21
ADDI R3, R1, 300    0     15    16    18    19    22
ADDI R6, R3, 1      0     15    19    21    22    23
ADDI R5, R1, 400    0     15    21    23    24    25
LW R3, R3, 0        0     15    21    23    24    25
LW R5, R5, 0        0     22    23    26    27    28
ADD R7, R3, R5      0     22    23    29    30    31
SW R7, R6, 0        0     22    24    31    32    33
ADDI R1, R1, 1      0     22    24    26    27    33
BEQ R2, R1, EXIT    0     25    26    28    29    34
JMP Loop            0     25    26    27    28    34
ADDI R3, R1, 300    0     25    29    31    32    35
ADDI R6, R3, 1      0     25    32    34    35    36
ADDI R5, R1, 400    0     33    34    36    37    38
LW R3, R3, 0        0     33    34    36    37    38
LW R5, R5, 0        0     33    35    39    40    41
ADD R7, R3, R5      0     33    35    42    43    44
SW R7, R6, 0        0     36    37    44    45    46
ADDI R1, R1, 1      0     36    37    39    40    46
BEQ R2, R1, EXIT    0     36    39    41    42    47
JMP Loop            0     36    39    40    41    47
ADDI R3, R1, 300    0     40    42    44    45    48
ADDI R6, R3, 1      0     40    45    47    48    49
ADDI R5, R1, 400    0     40    47    49    50    51
LW R3, R3, 0        0     40    47    49    50    51
LW R5, R5, 0        0     48    49    52    53    54
ADD R7, R3, R5      0     48    49    55    56    57
SW R7, R6, 0        0     48    50    57    58    59
ADDI R1, R1, 1      0     48    50    52    53    59
BEQ R2, R1, EXIT    0     51    52    54    55    60
JMP Loop            1     51    52    53    54    -1
ADDI R3, R1, 300    1     51    55    57    58    -1
ADDI R6, R3, 1      1     51    58    60    -1    -1
ADDI R5, R1, 400    1     59    60    -1    -1    -1
LW R3, R3, 0        1     59    60    -1    -1    -1
LW R5, R5, 0        1     59    -1    -1    -1    -1
ADD R7, R3, R5      1     59    -1    -1    -1    -1
Number of commitedd instructions is: 41
Number of cycles: 60
IPC = 0.683333
Total Branch Count = 4
Branch flushed = 0
Branches = 4
Branch Miss Prediction = 1
Branch Miss Prediction Rate = 0.25
```

2) This test case is an assembly code of the following C++ code:
For (int i=0; i<2; i++)
{
A[i+1] = A[i]+B[i];
}
This test case shows a Loop that loads a value from the memory adds to it one and then stores it back in the next address in the array, when the i is equal to 2 then the rest of the instructions after the Branching are all flushed, because Branch gets committed and we jump directly to Exit.

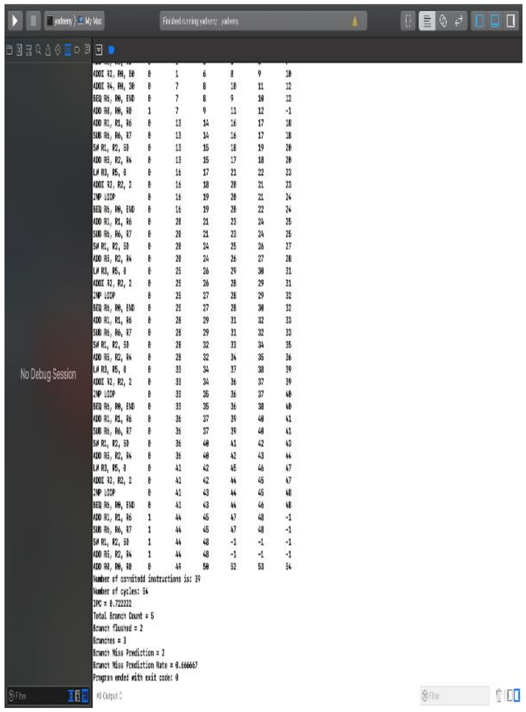
d. Test Case 4:
   1. Result:



2. Discussion

This testcase shows the use of JMP and RET, along with the use of labels instead of immediate. This example also shows a flushed instruction because of a RET instruction. The proceeding instruction was fetched, issued, executed and written. However, before it is committed, the instruction is flushed, because the RET instruction was committed and we jumped to its target address.

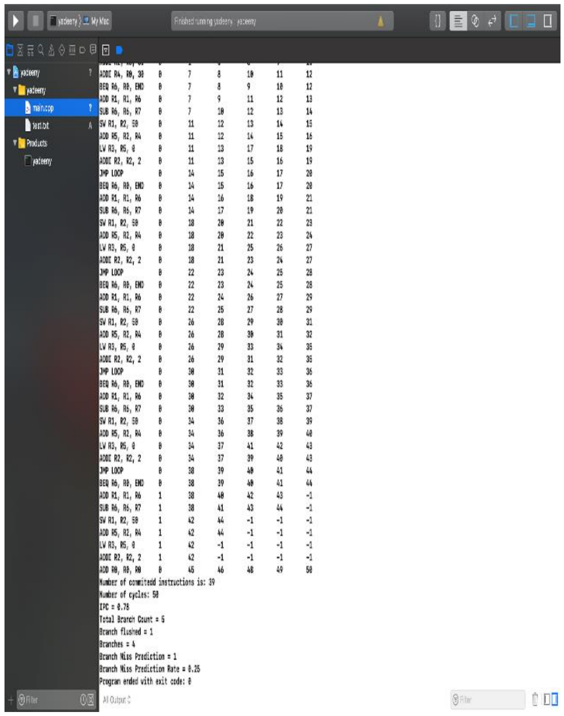
e. Testcase 5:
   1. Result with 2 bit branch prediction:



2. Discussion:

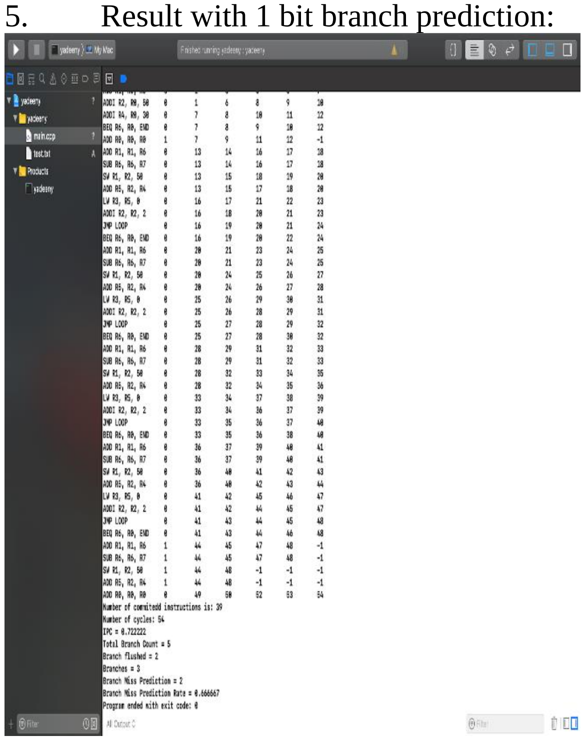
In the above figure, the branch prediction mode used was 2 bit prediction.
This testcase shows an assembly code that iterates several time in a loop. The branch instruction was repeated 5 times. The first time was predicted taken, however the true outcome was not taken. This is why the proceeding instruction was flushed before it commits when the branch was committing. The coming 3 instructions were predicted not taken and they were not taken, so no flushing was required. However, the last time, the prediction was not taken, but the true outcome was taken. This is why all the instructions that were fetched after the branch were flushed when the branch committed. There were no more instructions to fetch, so the program was finished.

3. Result with static branch prediction:



4. Discussion: In this example, all the branch immediates were positive. This is why they were all predicted not taken. Luckily, the first 4 occurences of the branch were really not taken, so the prediction was correct and no flushing was needed. However, the last branch occurrence was predicted not taken, but it was taken, so flushing was needed in this case.

5. Result with 1 bit branch prediction:

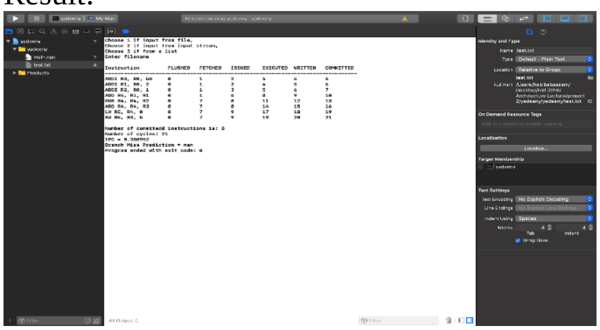


6. Discussion:

In the above figure, the branch prediction mode used was 1 bit prediction.
This testcase shows an assembly code that iterates several time in a loop. The branch instruction was repeated 5 times. The first time was predicted taken, however the true outcome was not taken. This is why the proceeding instruction was flushed before it commits when the branch was committing. The coming 3 instructions were predicted not taken and they were not taken, so no flushing was required. However, the last time, the prediction was not taken, but the true outcome was taken. This is why all the instructions that were fetched after the branch were flushed when the branch committed. There were no more instructions to fetch, so the program was finished.

f. Test Case 6:

1. Result:



2. Discussion:

The example above shows a load and store dependency, where then the store waited until load was committed in order to be able to execute the store. This example also shows 6 instructions that use the same reservation station and since there are only 3 reservation stations that hold these 4 instructions so we have a functional Unit hazard, so you can see that the first 2 are issued at the same time because its a dual issue and then once issued it issues the third instruction but not the fourth due to dependencies, so it waits for the second instruction to commit in order for the fourth instruction to get issued. After the first 4 instructions get committed then the next four are fetched.

g. Test Case 7:

1. Result:

Instruction	FLUSHED	FETCHED	ISSUED	EXECUTED	WRITTEN	COMMITTED
ADDI R1, R0, 10	0	1	2	4	5	6
ADD R2, R1, R3	0	1	2	7	8	9
SW R2, R1, 0	0	1	3	9	10	11
LW R2 , R1 , 0	0	1	3	12	13	14
SW R2, R1, 0	0	4	5	14	15	16
LW R3 , R1 , 0	0	4	5	17	18	19
BEQ R2, R3, JUMP	0	4	7	19	20	21
SUB R2, R1, R3	1	4	10	20	21	-1
LW R2 , R1 , 0	1	11	14	17	18	-1
SW R2, R1, 0	1	11	15	19	20	-1
ADD R2, R2, R2	1	11	17	20	21	-1
ADD R3, R2, R3	1	11	20	-1	-1	-1
LW R2 , R1 , 0	1	21	-1	-1	-1	-1
SW R2, R1, 0	1	21	-1	-1	-1	-1
ADD R3, R2, R3	0	22	23	25	26	27
LW R2 , R1 , 0	0	22	23	25	26	27
SW R2, R1, 0	0	22	24	27	28	29

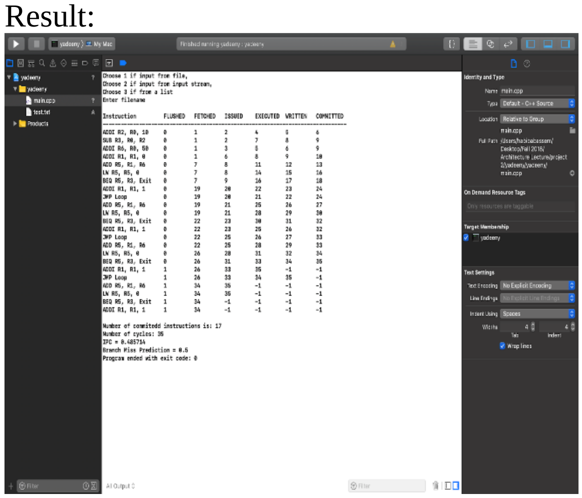
```
Number of commitedd instructions is: 10
Number of cycles: 29
IPC = 0.344828
Total Branch Count = 1
Branch flushed = 0
Branches = 1
Branch Miss Prediction = 1
Branch Miss Prediction Rate = 1
Program ended with exit code: 0
```

2. Discussion:

The example above tests 4 load and store instruction dependencies. At first the first 4 instructions are fetched, then since it is dual issue we issue the first two at the same time, but since instruction 2 uses the written back data from instruction one it stalls the execution until the first instruction is committed. The third and fourth instructions are lw and sw , they both get issued in the 3rd clock cycle but since sw is storing at the same memory location that lw wants to load from so it stalls until sw has been committed then lw executes, and since branch is taken everything after the branch is flushed until the Jump label which is where when branch is taken the address is supposed to jump to. The it continues normally where it lastly faces another lw and sw dependency.

h) Test Case 8:

1. Result:



```
Choose 1 if input from file,
Choose 2 if input from input stream,
Choose 3 if from a list
Enter filename
```

Instruction	FLUSHED	FETCHED	ISSUED	EXECUTED	WRITTEN	COMMITTED
ADDI R2, R0, 10	0	1	2	4	5	6
SUB R3, R0, R2	0	1	2	7	8	9
ADDI R6, R0, 50	0	1	3	5	6	9
ADDI R1, R1, 0	0	1	6	8	9	10
ADD R5, R1, R6	0	7	8	11	12	13
LW R5, R5, 0	0	7	8	14	15	16
BEQ R5, R3, Exit	0	7	9	16	17	18
ADDI R1, R1, 1	0	19	20	22	23	24
JMP Loop	0	19	20	21	22	24
ADD R5, R1, R6	0	19	21	25	26	27
LW R5, R5, 0	0	19	21	28	29	30
BEQ R5, R3, Exit	0	22	23	30	31	32
ADDI R1, R1, 1	0	22	23	25	26	32
JMP Loop	0	22	25	26	27	33
ADD R5, R1, R6	0	22	25	28	29	33
LW R5, R5, 0	0	26	28	31	32	34
BEQ R5, R3, Exit	0	26	31	33	34	35
ADDI R1, R1, 1	1	26	33	35	-1	-1
JMP Loop	1	26	33	34	35	-1
ADD R5, R1, R6	1	34	35	-1	-1	-1
LW R5, R5, 0	1	34	35	-1	-1	-1
BEQ R5, R3, Exit	1	34	-1	-1	-1	-1
ADDI R1, R1, 1	1	34	-1	-1	-1	-1

```
Number of commitedd instructions is: 17
Number of cycles: 35
IPC = 0.485714
Branch Miss Prediction = 0.5
Program ended with exit code: 0
```

2. Discussion:

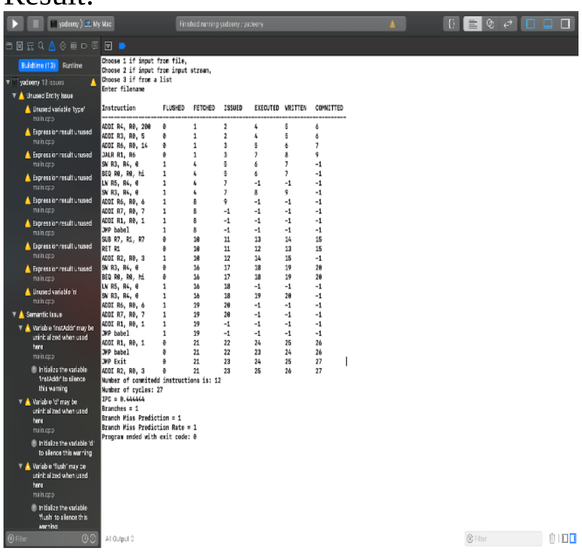
The test case above represents the following c++ code:

while (save[j]! = -k) j++;

In this test case the first four are fetched then the first two are issued, and the second instruction sub stalls waiting for Addi to commit in order to use the data in the register to perform the operation. The fourth and fifth instructions stalls due to the lack of space in the reservation station so it waits for the first and second instructions to leave the reservation station in order to get issued. Then in order for load to get issued it stalls waiting for add to get committed in order to use the value in the register being written into. Then it checks if the loaded value is equal to R3 , if it is it exits the loop jumps to exit else it increments by 1 and does the loop again until it jumps to exit. After that all the instructions after it are flushed.

h. Testcase 9:

1. Result:



2. Discussion:

In this example, there are 2 branch instructions. 1 of them is flushed so it is not counted in the total branch count, so we have 1 branch. This branch prediction mode is static, and the branch offset is positive, so it will be predicted not taken although it is actually taken. This is why we flush after it.

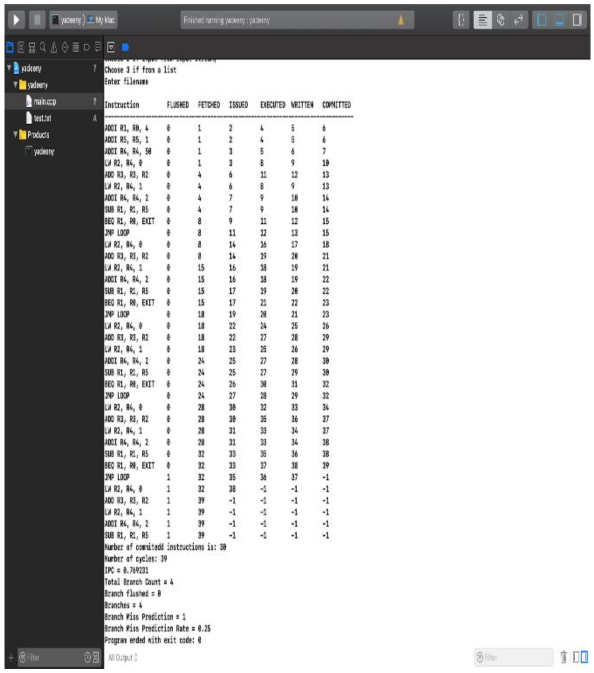
The first branch is flushed when the JALR R1, R6 is committed.

We flush again when RET R1 commits.

Finally, we branch because of the mispredicted branch when it commits.

i. Test Case 10:

1. Result for static branch prediction:

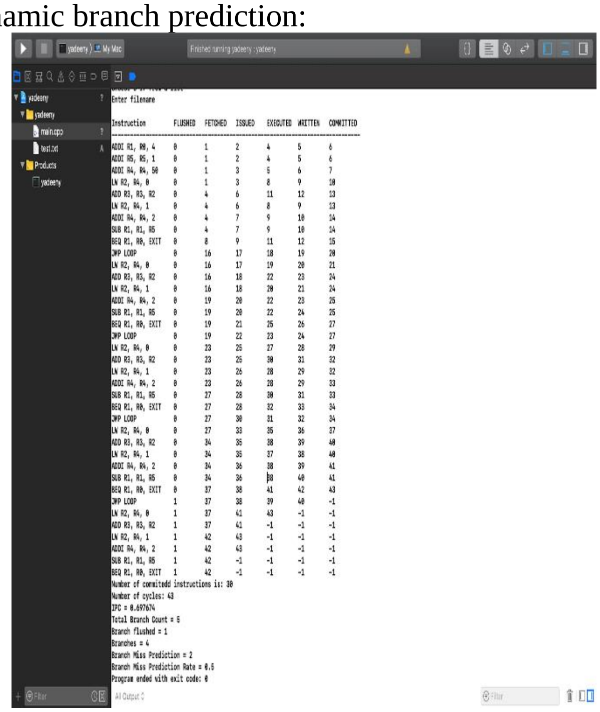


```
Choose 3 if from a list
Enter filename

Instruction        FLUSHED  FETCHED  ISSUED  EXECUTED  WRITTEN  COMMITTED
-------------------------------------------------------------------------
ADDI R1, R0, 4     0        1        2       4         5        6
ADDI R5, R5, 1     0        1        2       4         5        6
ADDI R4, R4, 50    0        1        3       5         6        7
LW R2, R4, 0       0        1        3       8         9        10
ADD R3, R3, R2     0        4        6       11        12       13
LW R2, R4, 1       0        4        6       8         9        13
ADDI R4, R4, 2     0        4        7       9         10       14
SUB R1, R1, R5     0        4        7       9         10       14
BEQ R1, R0, EXIT   0        8        9       11        12       15
JMP LOOP           0        8        11      12        13       15
LW R2, R4, 0       0        8        14      16        17       18
ADD R3, R3, R2     0        8        14      19        20       21
LW R2, R4, 1       0        15       16      18        19       21
ADDI R4, R4, 2     0        15       16      18        19       22
SUB R1, R1, R5     0        15       17      19        20       22
BEQ R1, R0, EXIT   0        15       17      21        22       23
JMP LOOP           0        18       19      20        21       23
LW R2, R4, 0       0        18       22      24        25       26
ADD R3, R3, R2     0        18       22      27        28       29
LW R2, R4, 1       0        18       23      25        26       29
ADDI R4, R4, 2     0        24       25      27        28       30
SUB R1, R1, R5     0        24       25      27        29       30
BEQ R1, R0, EXIT   0        24       26      30        31       32
JMP LOOP           0        24       27      28        29       32
LW R2, R4, 0       0        28       30      32        33       34
ADD R3, R3, R2     0        28       30      35        36       37
LW R2, R4, 1       0        28       31      33        34       37
ADDI R4, R4, 2     0        28       31      33        34       38
SUB R1, R1, R5     0        32       33      35        36       38
BEQ R1, R0, EXIT   0        32       33      37        38       39
JMP LOOP           1        32       35      36        37       -1
LW R2, R4, 0       1        32       38      -1        -1       -1
ADD R3, R3, R2     1        39       -1      -1        -1       -1
LW R2, R4, 1       1        39       -1      -1        -1       -1
ADDI R4, R4, 2     1        39       -1      -1        -1       -1
SUB R1, R1, R5     1        39       -1      -1        -1       -1
Number of commitedd instructions is: 30
Number of cycles: 39
IPC = 0.769231
Total Branch Count = 4
Branch flushed = 0
Branches = 4
Branch Miss Prediction = 1
Branch Miss Prediction Rate = 0.25
Program ended with exit code: 0
```

2. Discussion:

This example shows the iteration of a loop. There were 4 branches whose offsets are positive, so all their predictions were not taken. They were all not taken except the last one which was taken, this is why there is 1 misprediction.

3. Result for dynamic branch prediction:



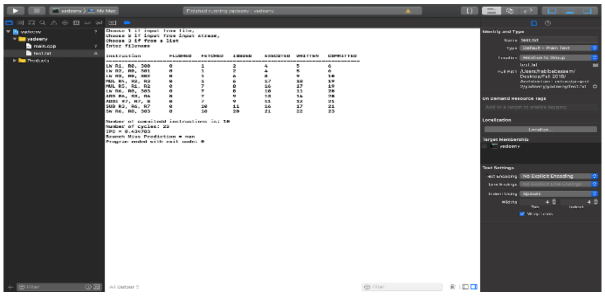
```
Enter filename

Instruction        FLUSHED  FETCHED  ISSUED  EXECUTED  WRITTEN  COMMITTED
-------------------------------------------------------------------------
ADDI R1, R0, 4     0        1        2       4         5        6
ADDI R5, R5, 1     0        1        2       4         5        6
ADDI R4, R4, 50    0        1        3       5         6        7
LW R2, R4, 0       0        1        3       8         9        10
ADD R3, R3, R2     0        4        6       11        12       13
LW R2, R4, 1       0        4        6       8         9        13
ADDI R4, R4, 2     0        4        7       9         10       14
SUB R1, R1, R5     0        4        7       9         10       14
BEQ R1, R0, EXIT   0        8        9       11        12       15
JMP LOOP           0        16       17      18        19       20
LW R2, R4, 0       0        16       17      19        20       21
ADD R3, R3, R2     0        16       18      22        23       24
LW R2, R4, 1       0        16       18      20        21       24
ADDI R4, R4, 2     0        19       20      22        23       25
SUB R1, R1, R5     0        19       20      22        24       25
BEQ R1, R0, EXIT   0        19       21      25        26       27
JMP LOOP           0        19       22      23        24       27
LW R2, R4, 0       0        23       25      27        28       29
ADD R3, R3, R2     0        23       25      30        31       32
LW R2, R4, 1       0        23       26      28        29       32
ADDI R4, R4, 2     0        23       26      28        29       33
SUB R1, R1, R5     0        27       28      30        31       33
BEQ R1, R0, EXIT   0        27       28      32        33       34
JMP LOOP           0        27       30      31        32       34
LW R2, R4, 0       0        27       33      35        36       37
ADD R3, R3, R2     0        34       35      38        39       40
LW R2, R4, 1       0        34       35      37        38       40
ADDI R4, R4, 2     0        34       36      38        39       41
SUB R1, R1, R5     0        34       36      38        40       41
BEQ R1, R0, EXIT   0        37       38      41        42       43
JMP LOOP           1        37       38      39        40       -1
LW R2, R4, 0       1        37       41      43        -1       -1
ADD R3, R3, R2     1        37       41      -1        -1       -1
LW R2, R4, 1       1        42       43      -1        -1       -1
ADDI R4, R4, 2     1        42       43      -1        -1       -1
SUB R1, R1, R5     1        42       -1      -1        -1       -1
BEQ R1, R0, EXIT   1        42       -1      -1        -1       -1
Number of commitedd instructions is: 30
Number of cycles: 43
IPC = 0.697674
Total Branch Count = 5
Branch flushed = 1
Branches = 4
Branch Miss Prediction = 2
Branch Miss Prediction Rate = 0.5
Program ended with exit code: 0
```

4. Discussion:

In this example, the first branch was predicted taken although it is not taken, then all the branches after were predicted not taken (in both 1 and 2 bit predictors). This is correct for all the branches except the last one because it is taken. Therefore the mispredictions are 2 (first and last 2 branches)

j. Testcase 11:

1. Result:

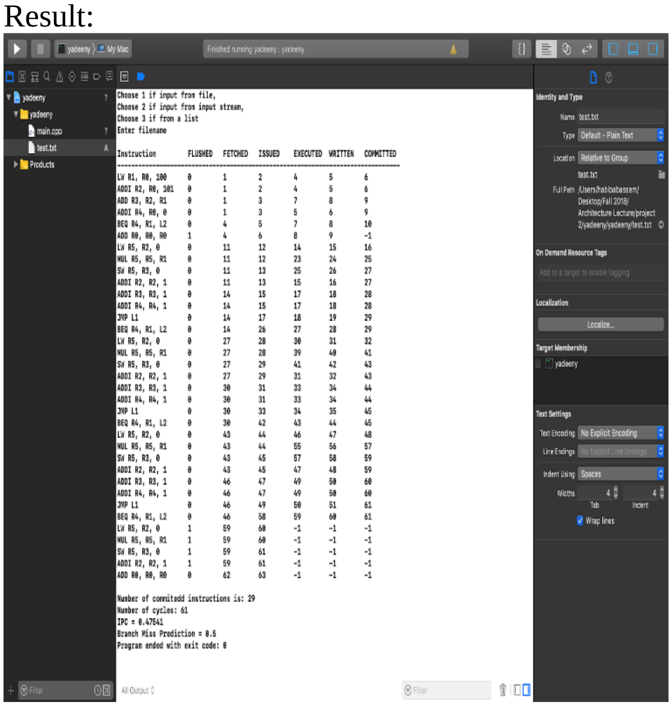


2. Discussion:

The differences between our result and the supplied testcase (sample 1) are all justified. As mentioned above (due to the cache structure), we fetch 4 instructions at a time. Also, for the third instruction, the load station was emptied in clock cycle 5. It made sense to issue the instruction into the reservation station in the following clock cycle, not in the same one. (I3 issued at clock cycle 6 as a result of that). All the differences between the results obtained are caused by these differences.

k. Testcase 12:

1. Result:

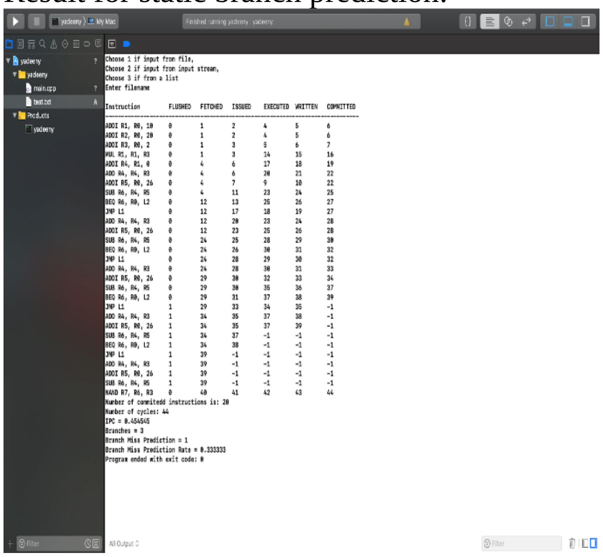


2. Discussion:

Similar to testcase 11, there are some slight differences in the assumptions made. This is because we fetch 4 instructions at a time when the instruction buffer is empty, as well as wait a cycle till the stations are actually emptied.

l. Testcase 13:

1. Result for static branch prediction:

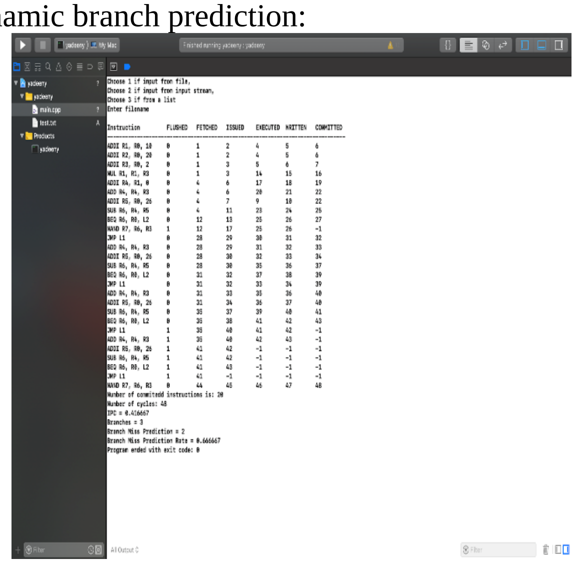


```
Choose 1 if input from file,
Choose 2 if input from input stream,
Choose 3 if from a list
Enter filename

Instruction        FLUSHED   FETCHED   ISSUED    EXECUTED  WRITTEN   COMMITTED
------------------------------------------------------------------------------
ADDI R1, R0, 10    0         1         2         4         5         6
ADDI R2, R0, 20    0         1         2         4         5         6
ADDI R3, R0, 2     0         1         3         5         6         7
MUL R1, R1, R3     0         1         3         14        15        16
ADDI R4, R1, 0     0         4         6         17        18        19
ADD R4, R4, R3     0         4         6         20        21        22
ADDI R5, R0, 26    0         4         7         9         10        22
SUB R6, R4, R5     0         4         11        23        24        25
BEQ R6, R0, L2     0         12        13        25        26        27
JMP L1             0         12        17        18        19        27
ADD R4, R4, R3     0         12        20        23        24        28
ADDI R5, R0, 26    0         12        23        25        26        28
SUB R6, R4, R5     0         24        25        28        29        30
BEQ R6, R0, L2     0         24        26        30        31        32
JMP L1             0         24        28        29        30        32
ADD R4, R4, R3     0         24        28        30        31        33
ADDI R5, R0, 26    0         29        30        32        33        34
SUB R6, R4, R5     0         29        30        35        36        37
BEQ R6, R0, L2     0         29        31        37        38        39
JMP L1             1         29        33        34        35        -1
ADD R4, R4, R3     1         34        35        37        38        -1
ADDI R5, R0, 26    1         34        35        37        39        -1
SUB R6, R4, R5     1         34        37        -1        -1        -1
BEQ R6, R0, L2     1         34        38        -1        -1        -1
JMP L1             1         39        -1        -1        -1        -1
ADD R4, R4, R3     1         39        -1        -1        -1        -1
ADDI R5, R0, 26    1         39        -1        -1        -1        -1
SUB R6, R4, R5     1         39        -1        -1        -1        -1
NAND R7, R6, R3    0         40        41        42        43        44
Number of commitedd instructions is: 20
Number of cycles: 44
IPC = 0.454545
Branches = 3
Branch Miss Prediction = 1
Branch Miss Prediction Rate = 0.333333
Program ended with exit code: 0
```

2. Discussion:

This example shows the iteration of a loop. The branch prediction here is static branch prediction. The branch offset is positive, so the prediction is not taken, which is a correct prediction except for the last branch. This is why the mispredicted branch is only 1 instruction.

3. Result for dynamic branch prediction:



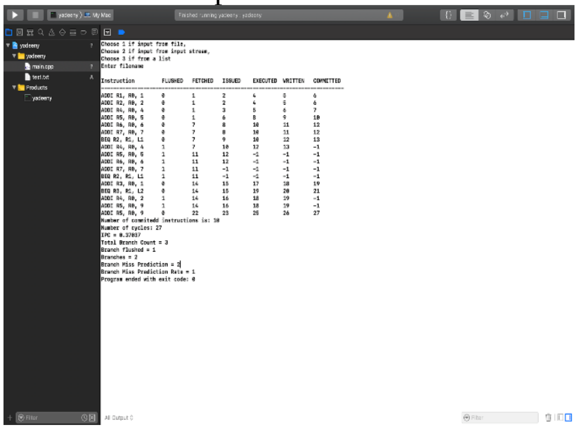
```
Choose 1 if input from file,
Choose 2 if input from input stream,
Choose 3 if from a list
Enter filename

Instruction        FLUSHED   FETCHED   ISSUED    EXECUTED  WRITTEN   COMMITTED
------------------------------------------------------------------------------
ADDI R1, R0, 10    0         1         2         4         5         6
ADDI R2, R0, 20    0         1         2         4         5         6
ADDI R3, R0, 2     0         1         3         5         6         7
MUL R1, R1, R3     0         1         3         14        15        16
ADDI R4, R1, 0     0         4         6         17        18        19
ADD R4, R4, R3     0         4         6         20        21        22
ADDI R5, R0, 26    0         4         7         9         10        22
SUB R6, R4, R5     0         4         11        23        24        25
BEQ R6, R0, L2     0         12        13        25        26        27
NAND R7, R6, R3    1         12        17        25        26        -1
JMP L1             0         28        29        30        31        32
ADD R4, R4, R3     0         28        29        31        32        33
ADDI R5, R0, 26    0         28        30        32        33        34
SUB R6, R4, R5     0         28        30        35        36        37
BEQ R6, R0, L2     0         31        32        37        38        39
JMP L1             0         31        32        33        34        39
ADD R4, R4, R3     0         31        33        35        36        40
ADDI R5, R0, 26    0         31        34        36        37        40
SUB R6, R4, R5     0         35        37        39        40        41
BEQ R6, R0, L2     0         35        38        41        42        43
JMP L1             1         35        40        41        42        -1
ADD R4, R4, R3     1         35        40        42        43        -1
ADDI R5, R0, 26    1         41        42        -1        -1        -1
SUB R6, R4, R5     1         41        42        -1        -1        -1
BEQ R6, R0, L2     1         41        43        -1        -1        -1
JMP L1             1         41        -1        -1        -1        -1
NAND R7, R6, R3    0         44        45        46        47        48
Number of commitedd instructions is: 20
Number of cycles: 48
IPC = 0.416667
Branches = 3
Branch Miss Prediction = 2
Branch Miss Prediction Rate = 0.666667
Program ended with exit code: 0
```

4. Discussion:

This example shows the iteration of a loop. The branch prediction here is dynamic branch prediction. The initial branch prediction is taken, but it is not taken. For the branch in the middle, the prediction is not taken and it is indeed not taken. However, for the last branch the prediction is not taken, and the branch is taken.

m. Testcase 14:

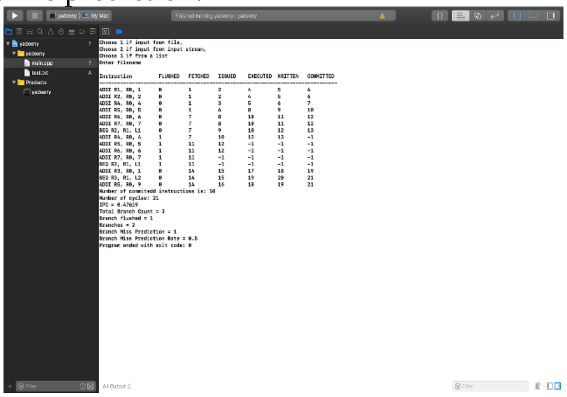
1. Result for static prediction:



2. Discussion:

This example shows branches that will be flushed because of other branches. A counter is kept of these flushed branch instructions because they should not count for the total number of branches.
In this example the branch offsets were positive, so they were predicted not taken, however both branches were taken. So the misprediction rate is 100%

3. Result for dynamic prediction:



4. Discussion:

In this example, the branch prediction is dynamic. The initial branch prediction is taken. Again, there is one flushed branch and this was not counted in the total number of branches.
The first branch was predicted taken, and it is taken so the prediction is correct. However, the second branch was predicted taken, but it is not taken.

n. Testcase 15:

1. Result for 1 bit prediction:

```
C:\WINDOWS\system32\cmd.exe
BEQ R2, R3, L2      0         849       850       851       852       853
ADDI R2, R0, 2      1         849       850       852       853       -1
ADDI R1, R1, 1      1         849       852       -1        -1        -1
JMP L1              1         849       852       853       -1        -1
BEQ R1, R6, EXIT    1         853       -1        -1        -1        -1
BEQ R2, R3, L2      1         853       -1        -1        -1        -1
ADDI R2, R0, 2      1         853       -1        -1        -1        -1
ADDI R1, R1, 1      1         853       -1        -1        -1        -1
ADDI R2, R0, 3      0         854       855       857       858       859
JMP L3              0         854       855       856       857       859
ADDI R1, R1, 1      0         854       856       858       859       860
JMP L1              0         854       856       857       858       860
BEQ R1, R6, EXIT    0         857       858       860       861       862
BEQ R2, R3, L2      1         857       858       859       860       -1
ADDI R2, R0, 3      1         857       860       862       -1        -1
JMP L3              1         857       860       861       862       -1
ADDI R1, R1, 1      1         861       862       -1        -1        -1
JMP L1              1         861       862       -1        -1        -1
BEQ R1, R6, EXIT    1         861       -1        -1        -1        -1
BEQ R2, R3, L2      1         861       -1        -1        -1        -1
ADDI R7, R0, 7      0         863       864       866       867       868
Number of commitedd instructions is: 556
Number of cycles: 868
IPC = 0.640553
Total Branch Count = 402
Branch flushed = 200
Branches = 202
Branch Miss Prediction = 101
Branch Miss Prediction Rate = 0.5
Press any key to continue . . .
```

2. Discussion: This testcase was designed to test the difference between 1 and 2 bit prediction. Instruction (BEQ R2, R3, L2) keeps alternating between taken and not taken states. This helps underline the difference between the two prediction methods. Note that in the above screenshot the branch misprediction rate was 50%. The code in this testcase essentially jumps once to a location that would cause a branch to be taken, and the next time to a place that causes a branch not to be taken. All of this is done while a counter is not yet decremented to zero.

3. Result for 2 bit prediction:

```
C:\WINDOWS\system32\cmd.exe
JMP L1              0          652        653        654        655        656
BEQ R1, R6, EXIT    0          652        653        655        656        657
BEQ R2, R3, L2      0          652        654        655        656        657
ADDI R2, R0, 2      1          652        654        656        657        -1
ADDI R1, R1, 1      1          655        656        -1         -1         -1
JMP L1              1          655        656        657        -1         -1
BEQ R1, R6, EXIT    1          655        657        -1         -1         -1
BEQ R2, R3, L2      1          655        -1         -1         -1         -1
ADDI R2, R0, 3      0          658        659        661        662        663
JMP L3              0          658        659        660        661        663
ADDI R1, R1, 1      0          658        660        662        663        664
JMP L1              0          658        660        661        662        664
BEQ R1, R6, EXIT    0          661        662        664        665        666
BEQ R2, R3, L2      1          661        662        663        664        -1
ADDI R2, R0, 2      1          661        664        666        -1         -1
ADDI R1, R1, 1      1          661        664        666        -1         -1
JMP L1              1          665        666        -1         -1         -1
BEQ R1, R6, EXIT    1          665        666        -1         -1         -1
BEQ R2, R3, L2      1          665        -1         -1         -1         -1
ADDI R2, R0, 2      1          665        -1         -1         -1         -1
ADDI R7, R0, 7      0          667        668        670        671        672
Number of commitedd instructions is: 556
Number of cycles: 672
IPC = 0.827381
Total Branch Count = 304
Branch flushed = 102
Branches = 202
Branch Miss Prediction = 52
Branch Miss Prediction Rate = 0.257426
Press any key to continue . . .
```

4. Discussion:

After using a 2 bit predictor the misprediction rate dropped significantly to 25.7%. This is because a mis predicted branch does not immediately change the prediction result, instead the prediction changes states in two increments in case it was originally in a strongly taken or strongly not taken state.

6. Branch Prediction Strategies Comparison:

The static branch prediction produced good results in some of the testcases mentioned above. However, this is only because most of the branch offsets were positive and the branch outcome was taken, or vice versa. However, in general, the static branch prediction is the worst when compared to 1 bit dynamic branch prediction and 2 bit dynamic branch prediction because it does not depend of the history of the branch.

1 bit branch predictor is a middle ground. It is better than static branch prediction and worse than 2 bit branch predictor. This is because it depends on the history of the branch (unlike the static branch prediction), but it is only composed of 2 states (unlike the 4 states of the 2 bit predictor).

In testcase 15, it shows an example when 1 bit predictor is way better than 2 bit predictor. This is when a branch outcome keeps shuffling between taken and not taken. The 1 bit predictor will predict wrong 100% of the time for the shuffling branch (50% overall because there are 2 branch instructions per iteration, one of which does not shuffle between taken and not taken). But 2-bit predictor will only predict wrong 50% of the time (25% overall because there are 2 branch instructions per iteration, one of which does not shuffle between taken and not taken).

7. Appendix

Test cases:

Test Case 1:

```
.text:
@8
ADDI R1, R2, 50
ADDI R1, R1, 3
LW R4, R1, 0
MUL R2, R4, R3
SW R2, R1, 0
BEQ R2, R5, 12
ADDI R2, R2, R1
.data:
@50 12
@53 13
@70 14
```

Test Case 2:

```
For(int i=0;i<5;i++)
{
A[i+1]=A[i]+!;
}

.text:
@8
ADDI R1, R1, 20
ADDI R2, R2, 0
Loop:
ADD R3, R4, R2
LW R3, R3, 0
ADDI R4, R3, 1
ADDI R6, R2, 4
ADD R3, R4, R6
SW R4, 0(R3)
ADDI R2, R2, 4
BEQ R2, R1, EXIT
JMP Loop
EXIT:
.data:
@0 12
@4 17
@8 18
```

@10 36
@17 82

Test Case 3:

```
For(int i=0;i<4;i++)
{
A[i]=A[i+!]+B[i];
}

.text:
@8
ADDI R1, R0, 20
ADDI R2, R0, 0
ADDI R4, R0, 0
Loop:
ADD R3, R4, R2
ADD R8, R5, R2
LW R3, R3, 0
ADDI R3, R3, 1
ADDI R6, R2, 4
LW R7, R8, 0
ADD R3, R7, R3
ADD R7, R4, R6
SW R3, R8, 0
ADDI R2, R2, 4
BEQ R2, R1, EXIT
JMP Loop
EXIT:
.data:
@0 12
@4 17
@36 18
@60 36
@72 82
```

Test Case 4:

```
.text:
@0
ADDI R2, R2, 10
ADD R2, R2, R3
SUB R3, R3, R2
NAND R5, R3, R1
JMP TRYIT
JMP EXIT
TRYIT:
```

```
MUL R3, R5, R3
ADDI R7, R0, 5
RET R7
EXIT:
ADD R0, R0, R0
```

Test Case 5:

```
.text:
@0
ADDI R6, R0, 4
ADDI R7, R0, 1
ADD R1, R0, R0
ADDI R2, R0, 50
ADDI R4, R0, 30
LOOP:
BEQ R6, R0, END
ADD R1, R1, R6
SUB R6, R6, R7
SW R1, R2, 50
ADD R5, R2, R4
LW R3, R5, 0
ADDI R2, R2, 2
JMP LOOP
END:
ADD R0, R0, R0
.data:
@80 30
@82 32
@84 34
@86 36
```

Test Case 6:

```
.text:
@0
ADDI R1, R0, 10
ADD R2, R1, R3
SW R2, R1, 0
LW R2 , R1 , 0
SW R2, R1, 0
LW R3 , R1 , 0
BEQ R2, R3, JUMP
SUB R2, R1, R3
LW R2 , R1 , 0
SW R2, R1, 0
ADD R2, R2, R2
```

```
JUMP:
ADD R3, R2, R3
LW R2 , R1 , 0
SW R2, R1, 0
```

Test Case 7:

X[5]= X[2*j-i];
Base address = r3
J = r1
I = r2
This code will load from memory[53] and store into memory[55]

```
.text:
@0
ADDI R3, R0, 50
ADDI R1, R0, 2
ADDI R2, R0, 1
ADD R4, R1, R1
SUB R4, R4, R2
ADD R4, R4, R3
LW R5, R4, 0
SW R5, R3, 5
.data:
@53 10
```

Test Case 8:

while (save[j] != -k) j++;

J = R1
K = R2 = 10
BASE ADDRESS = R6

```
.text:
@0
ADDI R2, R0, 10
SUB R3, R0, R2
ADDI R6, R0, 50
ADDI R1, R1, 0
Loop:
ADD R5, R1, R6
LW R5, R5, 0
BEQ R5, R3, Exit
ADDI R1, R1, 1
JMP Loop
Exit:
```

```
.data:
@50 2
@51 3
@52 -10
@53 7
```

Test Case 9:

```
.text:
@0
ADDI R4, R0, 200
ADDI R3, R0, 5
ADDI R6, R0, 14
JALR R1, R6
LAlel:
SW R3, R4, 0
BEQ R0, R0, hi
LW R5, R4, 0
label:
SW R3, R4, 0
ADDI R6, R0, 6
ADDI R7, R0, 7
hi:
ADDI R1, R0, 1
JMP babel
ADDI R2, R0, 2
babel:
JMP Exit
SUB R7, R1, R7
RET R1
Exit:
ADDI R2, R0, 3
.data:
@200 10
```

Test Case 10:

```
Int x = 50, i=0;
Do {
        Result += M[i];
        i++;
        result+=M[i];
        i++;
        x--;
}while (x!=0)
```

BASE ADDRESS -> R4
X -> R1
RESULT -> R3

```
.text:
@0
ADDI R1, R0, 4
ADDI R5, R5, 1
ADDI R4, R4, 50
LOOP:
LW R2, R4, 0
ADD R3, R3, R2
LW R2, R4, 1
ADDI R4, R4, 2
SUB R1, R1, R5
BEQ R1, R0, EXIT
JMP LOOP
EXIT:
.data:
@50 1
@51 2
@52 3
@53 4
@54 5
@55 6
@56 7
@57 8
@58 9
@59 10
@60 11
```

Test Case 11: (Dr's)

```
.text:
@0
LW R1, R0, 300
LW R2, R0, 301
LW R3, R0, 302
MUL R4, R2, R3
MUL R5, R1, R2
LW R6, R0, 303
ADD R6, R3, R6
ADDI R7, R7, 8
SUB R3, R6, R7
SW R6, R0, 303
.data:
@300 19
```

```
@301 2
@302 25
@303 208
```

Test Case 12: (Dr's)

```
.text:
@0
LW R1, R0, 100
ADDI R2, R0, 101
ADD R3, R2, R1
ADDI R4, R0, 0
L1:
BEQ R4, R1, L2
LW R5, R2, 0
MUL R5, R5, R1
SW R5, R3, 0
ADDI R2, R2, 1
ADDI R3, R3, 1
ADDI R4, R4, 1
JMP L1
L2:
ADD R0, R0, R0
.data:
@100 3
@101 12
@102 -5
@103 7
```

Test Case 13:

```
.text:
@0
ADDI R1, R0, 10
ADDI R2, R0, 20
BEQ R1, R2, L1:
ADDI R3, R0, 2
MUL R1, R1, R3
ADDI R4, R1, 0
L1:
ADD R4, R4, R3
ADDI R5, R0, 26
SUB R6, R4, R5
BEQ R6, R0, L2
JMP L1
ADDI R6, R0, 6
```

```
L2:
NAND R7, R6, R3
```

Testcase 14:

```
.text:
@0
ADDI R1, R0, 1
ADDI R2, R0, 2
L1:
ADDI R4, R0, 4
ADDI R5, R0, 5
ADDI R6, R0, 6
ADDI R7, R0, 7
BEQ R2, R1, L1
ADDI R3, R0, 1
BEQ R3, R1, L2
ADDI R4, R0, 2
L2:
ADDI R5, R0, 9
```

Testcase 15:

```
.text:
@0
ADDI R6, R0, 100
ADDI R1, R0, 0
ADDI R2, R0, 3
ADDI R3, R0, 3
L1:
BEQ R1, R6, EXIT
BEQ R2, R3, L2
ADDI R2, R0, 3
JMP L3
L2:
ADDI R2, R0, 2
L3:
ADDI R1, R1, 1
JMP L1
EXIT:
ADDI R7, R0, 7
```