INSTRUCTION PIPELINING (II)

1. Reducing Pipeline Branch Penalties

2. Instruction Fetch Units and Instruction Queues

3. Delayed Branching

4. Branch Prediction Strategies

5. Static Branch Prediction

6. Dynamic Branch Prediction

7. Branch History Table

Reducing Pipeline Branch Penalties

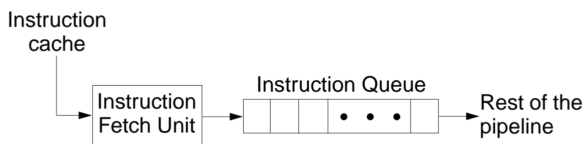
- Branch instructions can dramatically affect pipeline performance. Control operations (conditional and unconditional branch) are very frequent in current programs.

- Some statistics:
  - 20% - 35% of the instructions executed are branches (conditional and unconditional).
  - ~ 65% of the branches actually take the branch.
  - Conditional branches are much more frequent than unconditional ones (more than two times). More than 50% of conditional branches are taken.

- It is very important to reduce the penalties produced by branches.

Instruction Fetch Units and Instruction Queues

- Most processors employ sophisticated fetch units that fetch instructions before they are needed and store them in a queue.



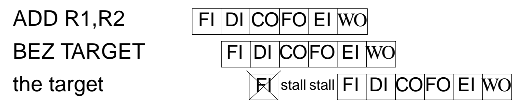
- The fetch unit also has the ability to recognize branch instructions and to generate the target address. Thus, penalty produced by *unconditional branches* can be drastically reduced: the fetch unit computes the target address and continues to fetch instructions from that address, which are sent to the queue. Thus, the rest of the pipeline gets a continuous stream of instructions, without stalling.
- The rate at which instructions can be read (from the instruction cache) must be sufficiently high to avoid an empty queue.
- With *conditional branches* penalties can not be avoided. The branch condition, which usually depends on the result of the preceding instruction, has to be known in order to determine the following instruction.

Delayed Branching

These are the pipeline sequences for a conditional branch instruction (see slide 12, lecture 3)

Branch is taken

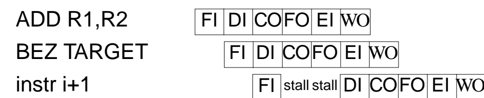
Clock cycle → 1 2 3 4 5 6 7 8 9 10 11 12



Penalty: 3 cycles

Branch is **not** taken

Clock cycle → 1 2 3 4 5 6 7 8 9 10 11 12



Penalty: 2 cycles

- The idea with delayed branching is to let the CPU do some useful work during some of the cycles which are shown above to be stalled.
- With delayed branching the CPU **always** executes the instruction that immediately follows after the branch and only then alters (if necessary) the sequence of execution. The instruction after the branch is said to be in the *branch delay slot*.

Delayed Branching (cont'd)

<u>This is what the programmer has written</u>

This instruction does not influence any of the instructions which follow until the branch; it also doesn't influence the outcome of the branch. → MUL R3,R4

```
MUL  R3,R4     R3 ← R3*R4
SUB  #1,R2     R2 ← R2-1
ADD  R1,R2     R1 ← R1+R2
BEZ  TAR       branch if zero
MOVE #10,R1    R1 ← 10
- - - - - - - - - - - - -
TAR  - - - - - - - - - - - - -
```

This instruction should be executed only if the branch is not taken. → MOVE #10,R1

- **The compiler** (assembler) has to find an instruction which can be moved from its original place into *the* branch delay slot after the branch and which will be executed regardless of the outcome of the branch.

<u>This is what the compiler (assembler) has produced and what actually will be executed</u>:

```
     SUB   #1,R2
     ADD   R1,R2
     BEZ   TAR
     MUL   R3,R4
     MOVE  #10,R1
     - - - - - - - - - - - - -
TAR  - - - - - - - - - - - - -
```

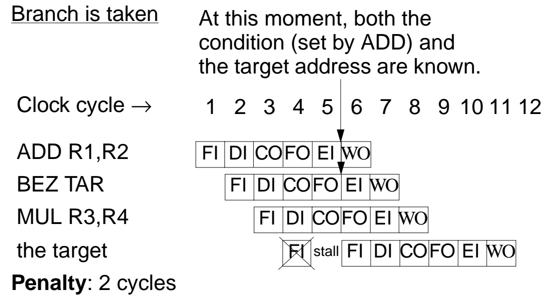
MUL R3,R4 ← This instruction will be executed regardless of the condition.

MOVE #10,R1 ← This will be executed only if the branch has not been taken

Delayed Branching (cont'd)

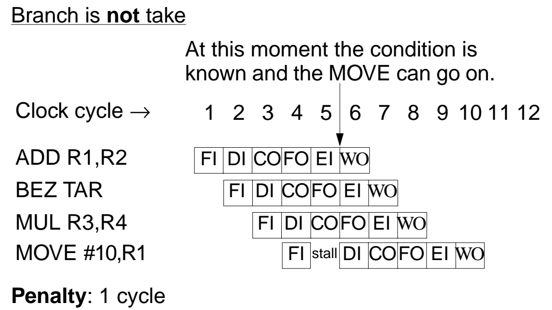
This happens in the pipeline:

<u>Branch is taken</u>



Penalty: 2 cycles

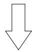
<u>Branch is **not** take</u>



Penalty: 1 cycle

Delayed Branching (cont'd)

- What happens if the compiler is not able to find an instruction to be moved after the branch, into the branch delay slot?



In this case a NOP instruction (an instruction that does nothing) has to be placed after the branch. In this case the penalty will be the same as without delayed branching.

```
     MUL   R2,R4
     SUB   #1,R2
     ADD   R1,R2
     BEZ   TAR
     NOP
     MOVE  #10,R1
     - - - - - - - - - - - - -
TAR  - - - - - - - - - - - - -
```

MUL **R2**,R4 ← Now, with R2, this instruction influences the following ones and cannot be moved from its place.

- Some statistics show that for between 60% and 85% of branches, sophisticated compilers are able to find an instruction to be moved into the branch delay slot.

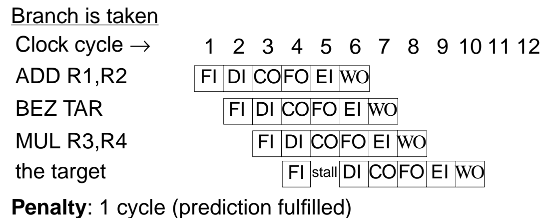
<u>Branch Prediction</u>

- In the last example we have considered that the *branch will not be taken* and we fetched the instruction following the branch; in the case the branch was taken the fetched instruction was discarded. As result, we had

  branch penalty of
  - 1 if the branch is **not** taken (prediction fulfilled)
  - 2 if the branch is taken (prediction not fulfilled)

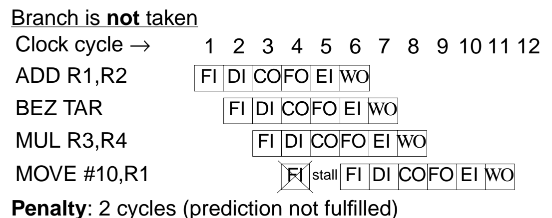
- Let us consider the opposite prediction: *branch taken*. For this solution it is needed that the target address is computed in advance by an instruction fetch unit.

<u>Branch is taken</u>



Penalty: 1 cycle (prediction fulfilled)

<u>Branch is **not** taken</u>



Penalty: 2 cycles (prediction not fulfilled)

Branch Prediction (cont'd)

- Correct branch prediction is very important and can produce substantial performance improvements.
- Based on the predicted outcome, the respective instruction can be fetched, as well as the instructions following it, and they can be placed into the instruction queue (see slide 3). If, after the branch condition is computed, it turns out that the prediction was correct, execution continues. On the other hand, if the prediction is not fulfilled, the fetched instruction(s) must be discarded and the correct instruction must be fetched.
- To take full advantage of branch prediction, we can have the instructions not only fetched but also begin execution. This is known as *speculative execution*.
- *Speculative execution* means that instructions are executed before the processor is certain that they are in the correct execution path. If it turns out that the prediction was correct, execution goes on without introducing any branch penalty. If, however, the prediction is not fulfilled, the instruction(s) started in advance and all their associated data must be purged and the state previous to their execution restored.
- Branch prediction strategies:
  1. Static prediction
  2. Dynamic prediction

Static Branch Prediction

- Static prediction techniques do not take into consideration execution history.

Static approaches:

- Predict never taken (Motorola 68020): assumes that the branch is not taken.
- Predict always taken: assumes that the branch is taken.
- Predict depending on the branch direction (PowerPC 601):
  - predict branch taken for backward branches;
  - predict branch not taken for forward branches.

Dynamic Branch Prediction

- Dynamic prediction techniques improve the accuracy of the prediction by recording the history of conditional branches.

One-Bit Prediction Scheme

- One-bit is used in order to record if the last execution resulted in a branch taken or not. The system predicts the same behavior as for the last time.

  Shortcoming

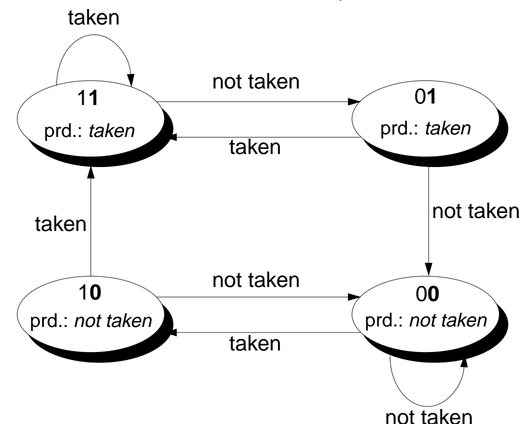
  When a branch is almost always taken, then when it is not taken, we will predict incorrectly twice, rather than once:

```
        -----------
LOOP    -----------
        -----------
        BNZ   LOOP
        -----------
```

- After the loop has been executed for the first time and left, it will be remembered that BNZ has **not** been taken. Now, when the loop is executed again, after the first iteration there will be a false prediction; following predictions are OK until the last iteration, when there will be a *second* false prediction.
- In this case the result is even worse than with static prediction considering that backward loops are always taken (PowerPC 601 approach).

Two-Bit Prediction Scheme

- With a two-bit scheme predictions can be made depending on the last two instances of execution.
- A typical scheme is to change the prediction only if there have been two incorrect predictions in a row.

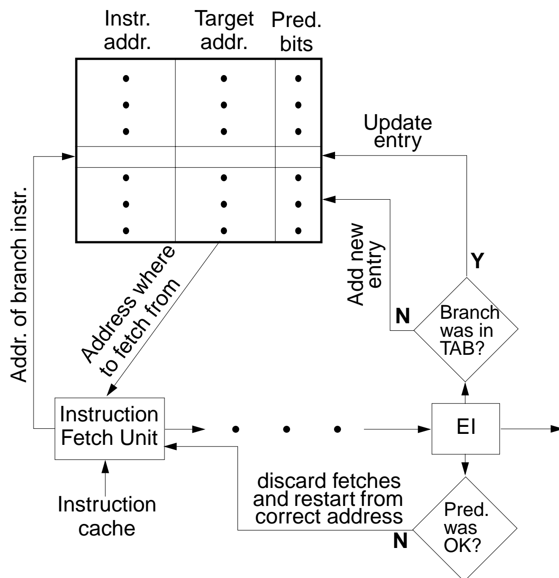



```
        -----------
LOOP    -----------
        -----------
        BNZ   LOOP
        -----------
```

After the first execution of the loop the bits attached to BNZ will be 01; now, there will be always one false prediction for the loop, at its exit.

Branch History Table

- History information can be used not only to predict the outcome of a conditional branch but also to avoid recalculation of the target address. Together with the bits used for prediction, the target address can be stored for later use in a *branch history table*.



Branch History Table (cont'd)

Some explanations to previous figure:

- *Address where to fetch from*: If the branch instruction is not in the table the next instruction (*address PC+1*) is to be fetched. If the branch instruction is in the table first of all a prediction based on the *prediction bits* is made. Depending on the prediction outcome the next instruction (*address PC+1*) or the instruction at the *target address* is to be fetched.
- *Update entry*: If the branch instruction has been in the table, the respective entry has to be updated to reflect the correct or incorrect prediction.
- *Add new entry*: If the branch instruction has not been in the table, it is added to the table with the corresponding information concerning branch outcome and target address. If needed one of the existing table entries is discarded. Replacement algorithms similar to those for cache memories are used.

- Using dynamic branch prediction with history tables up to 90% of predictions can be correct.
- Both Pentium and PowerPC 620 use speculative execution with dynamic branch prediction based on a branch history table.

Summary

- Branch instructions can dramatically affect pipeline performance. It is very important to reduce penalties produced by branches.
- Instruction fetch units are able to recognize branch instructions and generate the target address. Fetching at a high rate from the instruction cache and keeping the instruction queue loaded, it is possible to reduce the penalty for unconditional branches to zero. For conditional branches this is not possible because we have to wait for the outcome of the decision.
- Delayed branching is a compiler based technique aimed to reduce the branch penalty by moving instructions into the branch delay slot.
- Efficient reduction of the branch penalty for conditional branches needs a clever branch prediction strategy. Static branch prediction does not take into consideration execution history. Dynamic branch prediction is based on a record of the history of a conditional branch.
- Branch history tables are used to store both information on the outcome of branches and the target address of the respective branch.