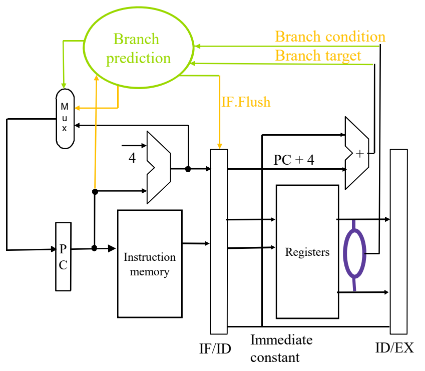
Dynamic branch prediction

- Goal
  - Predict the branch outcome (taken or not taken) and the branch target address (if taken, where should we go?) at run time.
  - In short, predict the next PC (target address or PC+4)?
  - Should be able to kill the predicted instructions if you discover that the prediction was wrong

- Can we dynamically predict, at rum time, the next PC when we are fetching from the current PC?

- Need to dynamically predict:
  - Branch condition
  - Branch target address

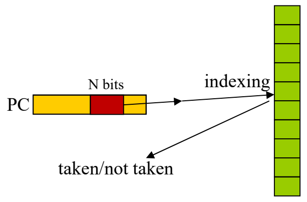


What to predict – T/NT

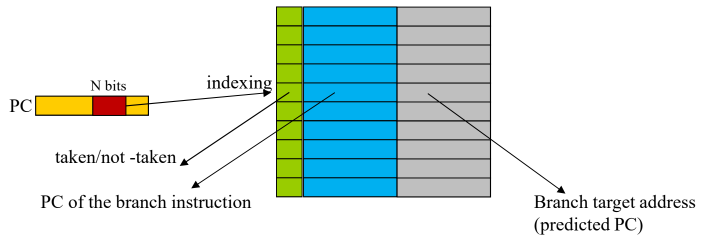
- **Let's first focus on predicting taken (T)/not taken (NT)**

- **Branch outcome may be biased for individual branches**
  - Example: in loops, branches may be mostly taken or mostly not taken

- **Dynamic prediction using a simple 1-bit predictor**
  - Remember the last behavior (taken/not-taken) using a hash table
    - 2^N entries – index using N bits from PC (for some N)
    - Hashing → may have collision
    - Collision → problem??
  - When should we check the table?
  - If prediction is wrong
    - Take corrective action
    - update the prediction buffer



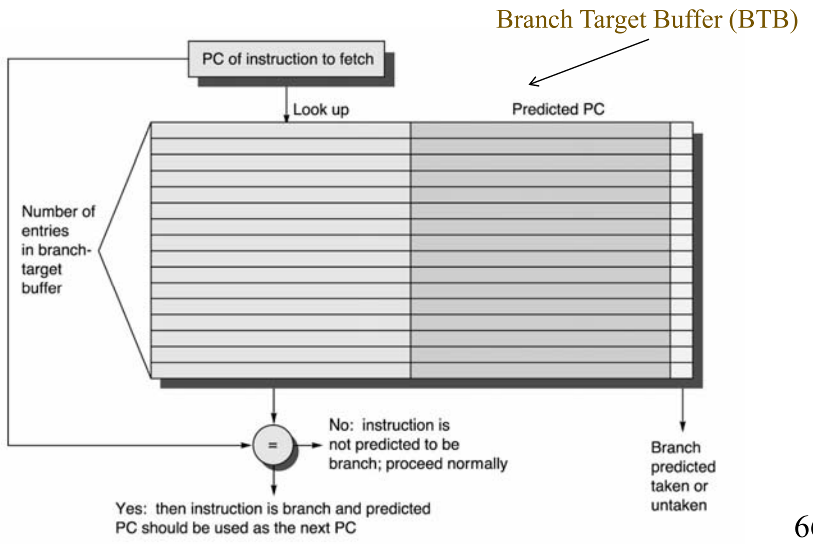
Target prediction (Branch Target Buffer – BTB)



- We would like to check the table while fetching the instruction
  - Are we sure that the BTB entry refers to a branch instruction?
  - Are we sure it refers to the correct branch instruction?
- Should store and match all bits of PC.
- Add the last branch address to the table (to predict the target)

Target prediction with Branch Target Buffer

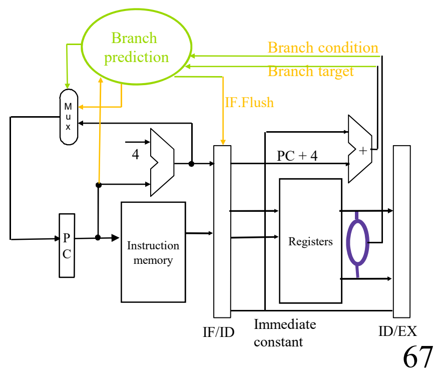
Records the target address for the last execution of the branch instruction


Branch Target Buffer (BTB)
PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be branch; proceed normally
Yes: then instruction is branch and predicted PC should be used as the next PC
Branch predicted taken or untaken

Target prediction with BTB

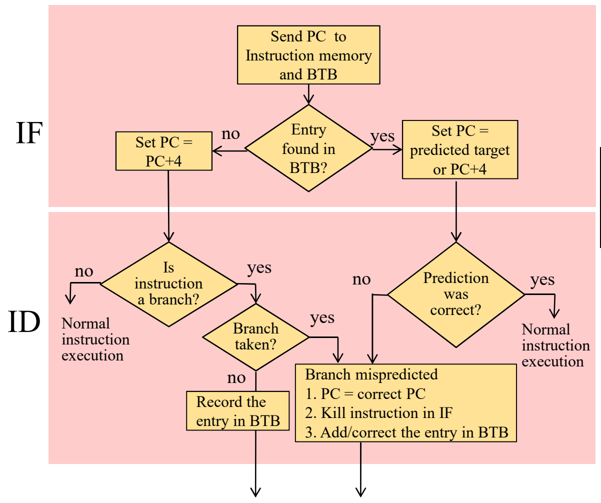
- If *can predict* and "*Prediction = taken*", then set next PC = stored target
- If *can predict* and "*Prediction = untaken*", then set next PC = PC+4
- If cannot predict, then use PC+4 (assume branch is not taken). This happens if either
  - the instruction is a branch, but no prediction is found in BTB
  - the instruction is not a branch

- After resolving a branch, if we find that it was mispredicted
  - Undo the damage (flush the wrong instruction)
  - Restore correct PC
  - Update BTB with new information

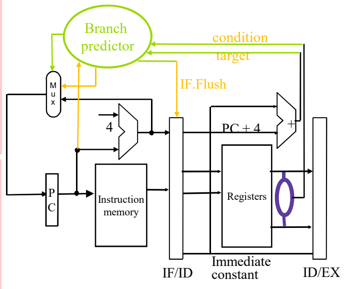


Branch prediction & pipelining

Assuming that branch condition and target are resolved in ID stage



IF
Send PC to Instruction memory and BTB
Entry found in BTB?
no
Set PC = PC+4
yes
Set PC = predicted target or PC+4
ID
Is instruction a branch?
no
Normal instruction execution
yes
Branch taken?
no
Record the entry in BTB
yes
Prediction was correct?
no
Branch mispredicted
1. PC = correct PC
2. Kill instruction in IF
3. Add/correct the entry in BTB
yes
Normal instruction execution



A similar chart may be drawn if branch condition/target are resolved in EX

Shortcoming of 1-bit branch predictors

- Remember the last behavior of the branch

```
for (i=0; i< 100; i++) {
  A[i] = B[i] * C[i];
  D[i] = E[i] / F[i];
}
```

this is a conditional branch

- How many prediction hits and misses? Prediction accuracy?

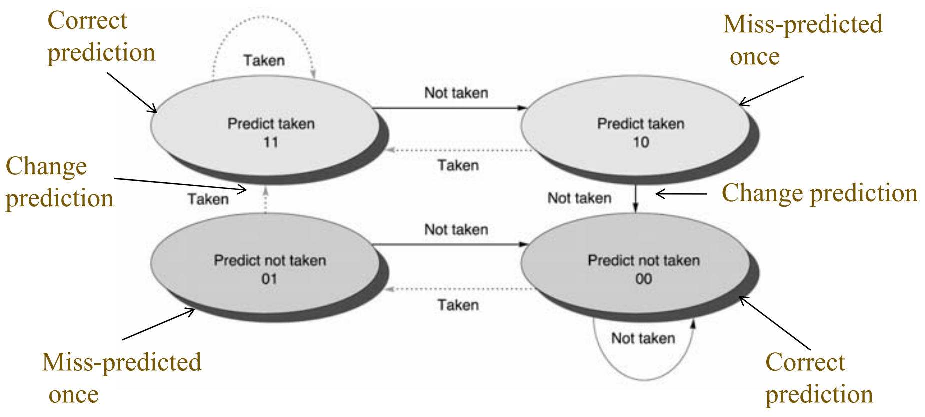
```
for (j=0; j<100; j++) {
  for (i=0; i< 5; i++) {
    A[i] = B[i] * C[i];
    D[i] = E[i] / F[i];
  }
}
```

Predicted	--	T	T	T	T	NT	T	T	T	T	NT	T	T	...
Actual	T	T	T	T	NT	T	T	T	T	NT	T	T	..	...

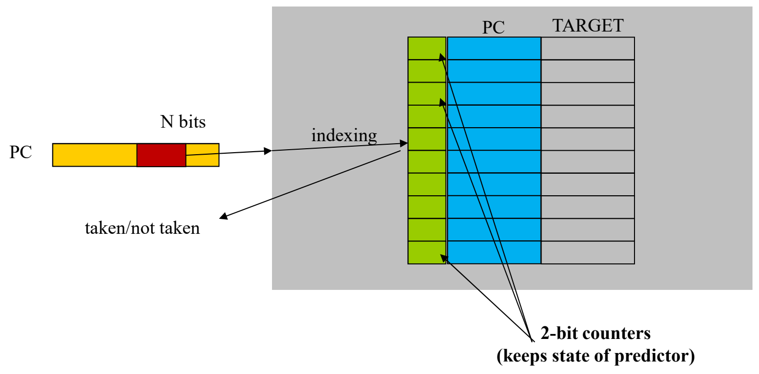
this branch is predicted wrong twice every inner loop invocation (every 5 branches)

2-bit predictor

- Requires two consecutive miss-predictions to flip direction



2-Bit Branch prediction buffer



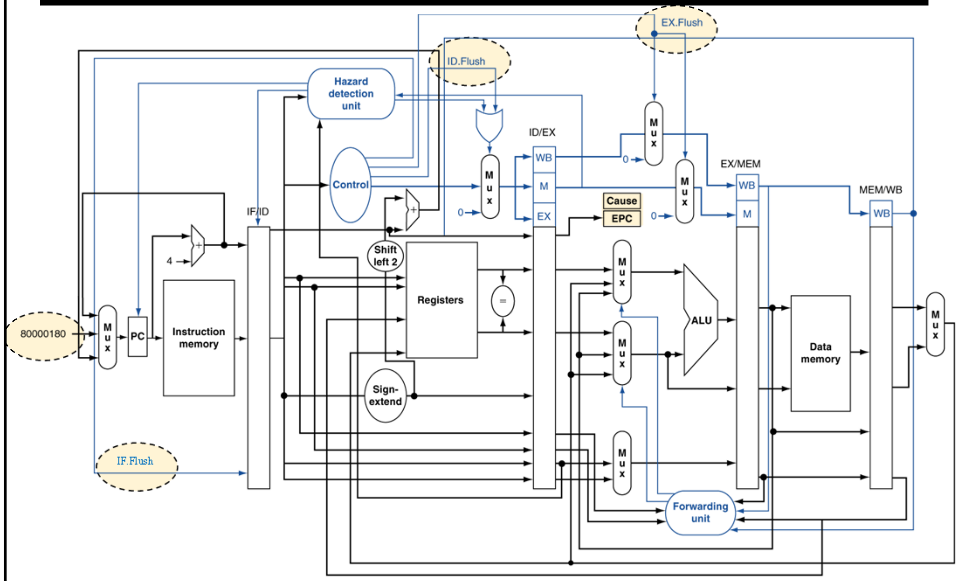
Handling exceptions (Section 4.9)

- Exceptions (traps or interrupts) are an important part of a processor
- Modern OS and debuggers depend on hardware support for exceptions
- Interrupt mechanism allows efficient use of CPU cycles by allowing the CPU to do useful work while waiting for an event.

- *Precise exception*
  - Execute all instructions before the faulting (trapped/interrupted) instruction
  - Do not execute the faulting instruction and the following ones
  - Resume from the faulting instruction after the exception is taken care of

- How does MIPS handle exceptions?
  - Record the PC of the next instruction in a special place (the *EPC* register)
  - Record the cause of the exception (the *Cause* register)
  - Squash the faulting instruction and the instructions following it
  - Jump to a pre-determined handling routine (PC = 8000 00180)
  - When done, the handling routine will execute a “return from interrupt” to resume from the instruction following the faulting instruction (EPC → PC), or the faulting instruction (EPC-4 → PC) – depending on implementation.

5-stage pipeline with Exceptions



The Handler

- Typically, the exception handler will determine the action(s) to be taken
  - If consequences are not fatal
    - take corrective action
    - use EPC to return to program
  - Otherwise
    - Report error using EPC, cause, …
    - Terminate program

- Handler jumps to an address which is determined by the cause of the exception
  - Addresses are stored in a vector which is indexed by the cause
  - Example:

Undefined opcode:	C000 0000
Overflow:	C000 0020
…:	C000 0040

For external interrupts: the handler address is stored in a vector which is indexed by the interrupt type. The PC+4 is saved in EPC and the handler's address replaces the current PC.