BRANCH PREDICTION TECHNIQUES

Dr Noor Mahammad Sk

Conditional Branch Instructions

- Conditional Branch Instruction: the branch is taken only if the condition is satisfied.
- The branch target address is stored in the Program Counter (PC) instead of the address of the next instruction in the sequential instruction stream.
- Examples of branches for MIPS processor: beq(branch on equal) and bne(branch on not equal)
- beq $s1, $s2, L1   # go to L1 if ($s1 == $s2)
- bne $s1, $s2, L1   # go to L1 if ($s1 != $s2)

Conditional branches for MIPS processor: I (Immediate) Format

op	rs	rt	address
6 bit	5 bit	5 bit	16 bit

- For conditional branches, the instruction fields are:
- op(opcode) identifies the instruction type;
- rs first register to compare;
- rt second register to compare;
- address(16-bit) indicates the word offset relative to the PC (PC-relative word address)
- The offset corresponding to the L1 label (Branch Target Address) is relative to the Program Counter (PC):
- (L1-PC) /4
- beq $s1, $s2, L1

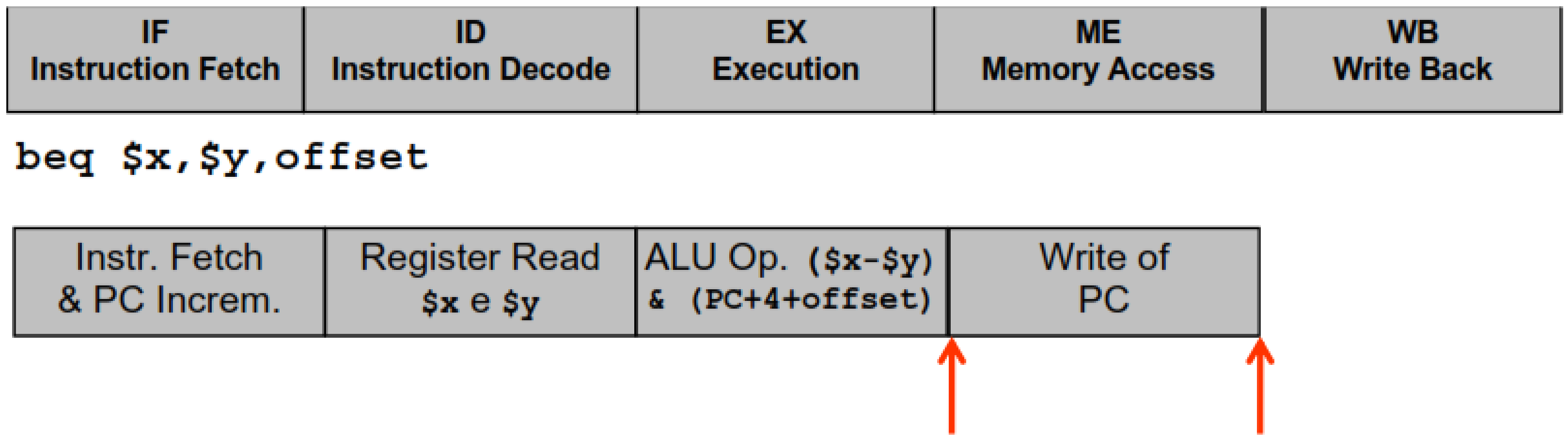
Execution of conditional branches for 5-stage MIPS pipeline

```
beq $x,$y,offset
```

Instr. Fetch & PC Increm.	Register Read `$x` e `$y`	ALU Op. `($x-$y)` `& (PC+4+offset)`	Write of PC

- Instruction fetch and PC increment
- Registers read **($x and $y) from Register File.**
- ALU operation to compare registers ($x and $y) to derive **Branch Outcome (branch taken or branch not taken).**
- Computation of **Branch Target Address (PC+4+offset): the value** (PC+4) is added to the least significant 16 bit of the instruction after sign extension
- The result of registers comparison from ALU (Branch Outcome) is used to decide the value to be stored in the PC: (PC+4) or (PC+4+offset).

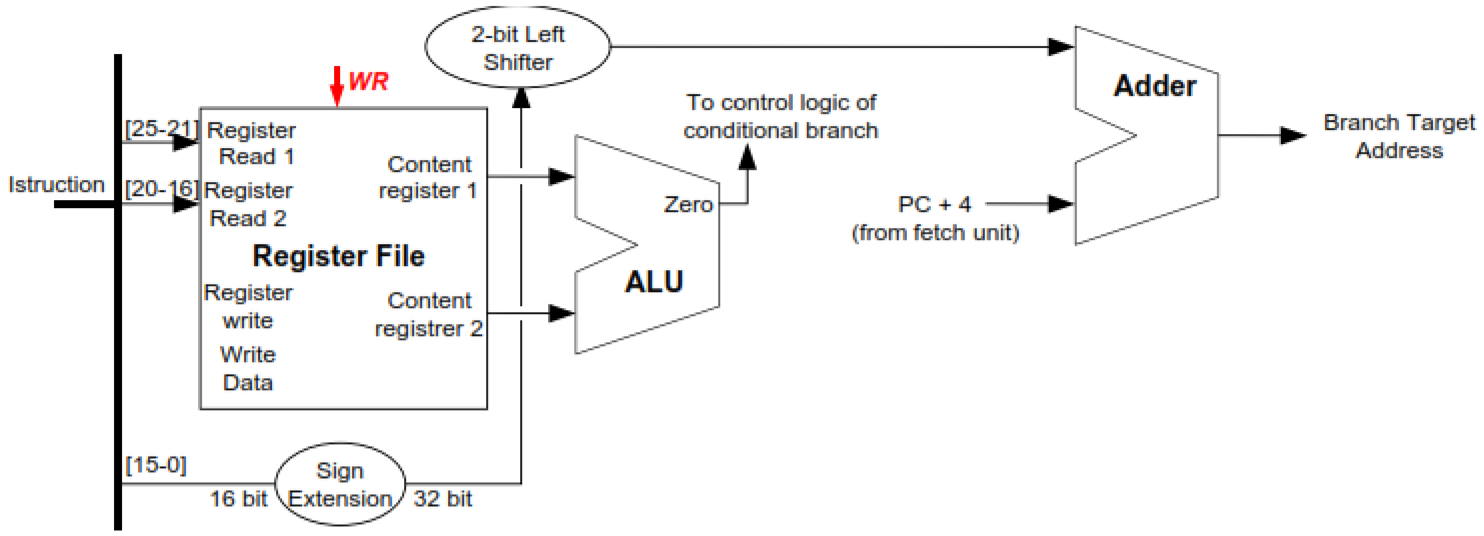
Execution of conditional branches for 5-stage MIPS pipeline



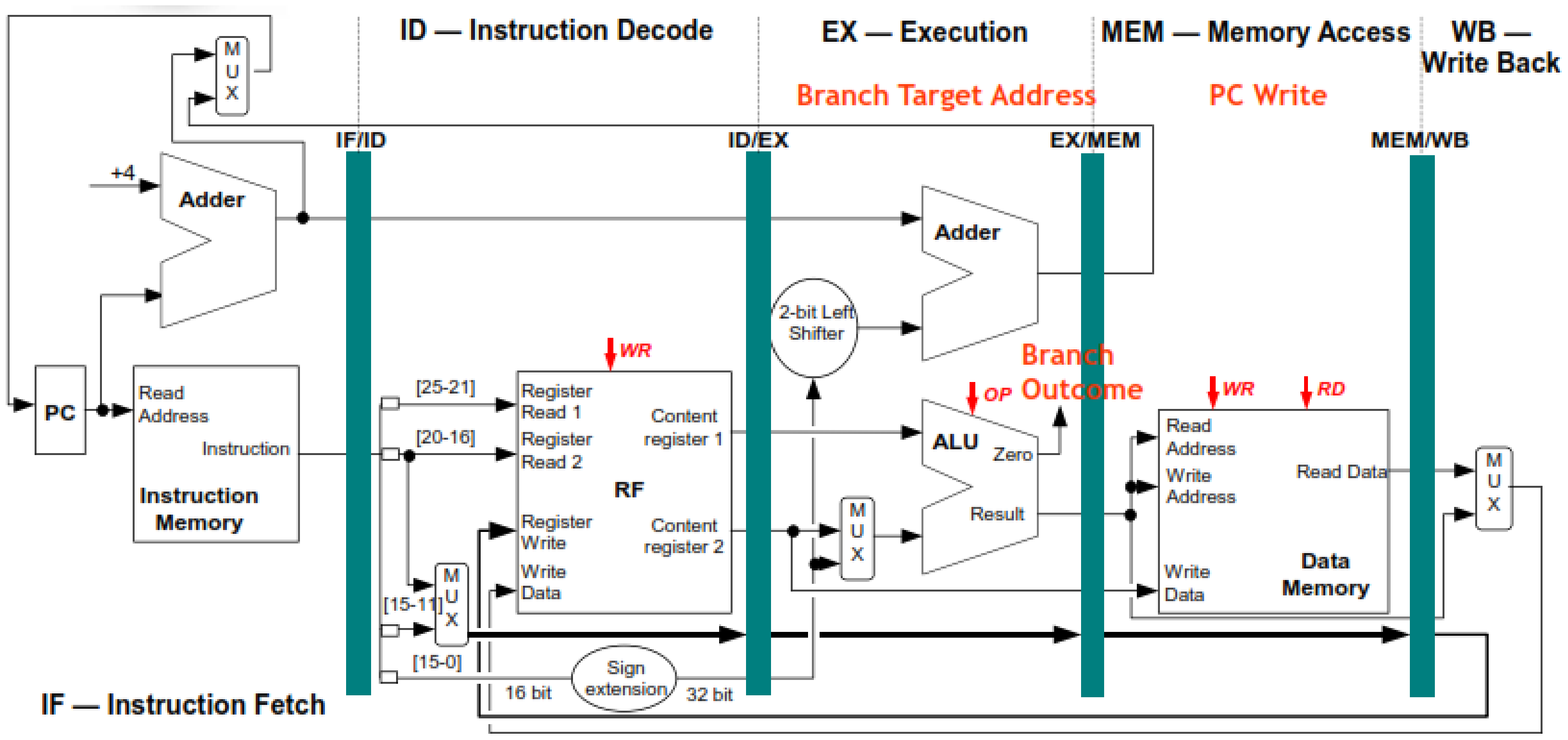
- **Branch Outcome and Branch Target Address are ready at** the end of the EX stage (3rd stage)
- Conditional branches are solved when **PC is updated at** the end of the ME stage (4th stage)

Execution of conditional branches for MIPS

- Processor resources to execute conditional branches:



Implementation of the 5-stage MIPS Pipeline



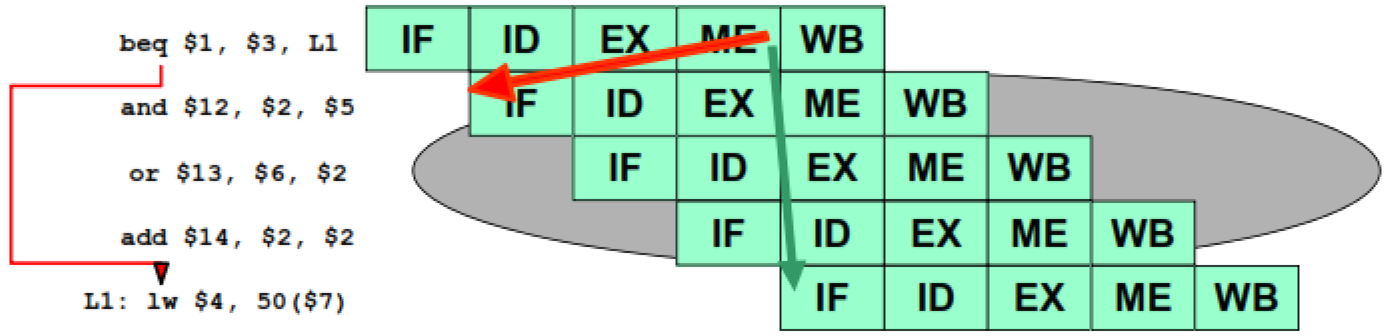
The Problem of Control Hazards

- **Control hazards: Attempt to make a decision on the** next instruction to fetch before the branch condition is evaluated.
- Control hazards arise from the pipelining of conditional branches and other instructions changing the PC.
- Control hazards reduce the performance from the ideal speedup gained by the pipelining since they can make it necessary to stall the pipeline.

Branch Hazards

- To feed the pipeline we need to fetch a new instruction at each clock cycle, but the branch decision (to change or not change the PC) is taken during the MEM stage.
- This delay to determine the correct instruction to fetch is called **Control Hazard or Conditional Branch Hazard**
- If a branch changes the PC to its target address, it is a **taken branch**
- If a branch falls through, it is **not taken or untaken.**

Branch Hazards: Example

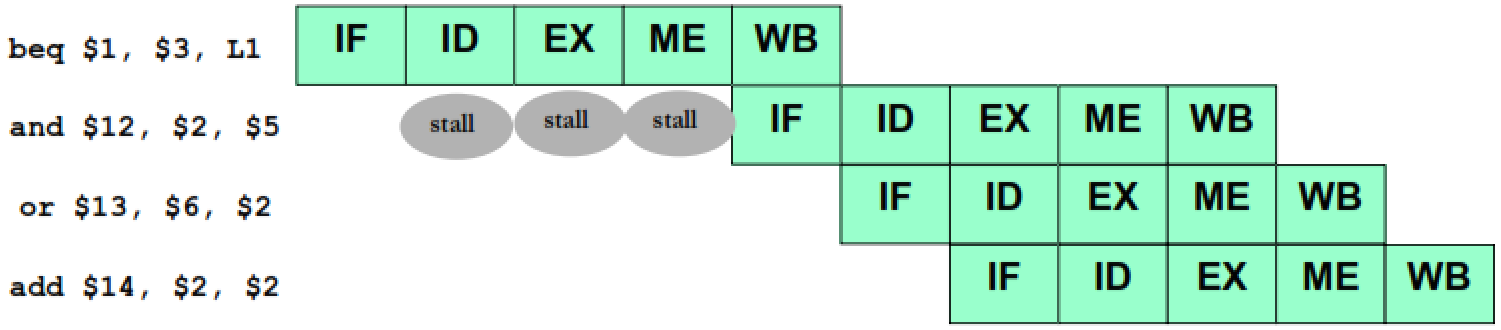


- The branch instruction may or may not change the PC in MEM stage, but the next 3 instructions are fetched and their execution is started
- If the branch is **not taken, the pipeline execution is OK**
- If the branch is **taken, it is necessary to *flush the next 3 instructions*** in the pipeline and fetched the **lw instruction at the branch target** address **(L1)**

Branch Hazards: Solutions

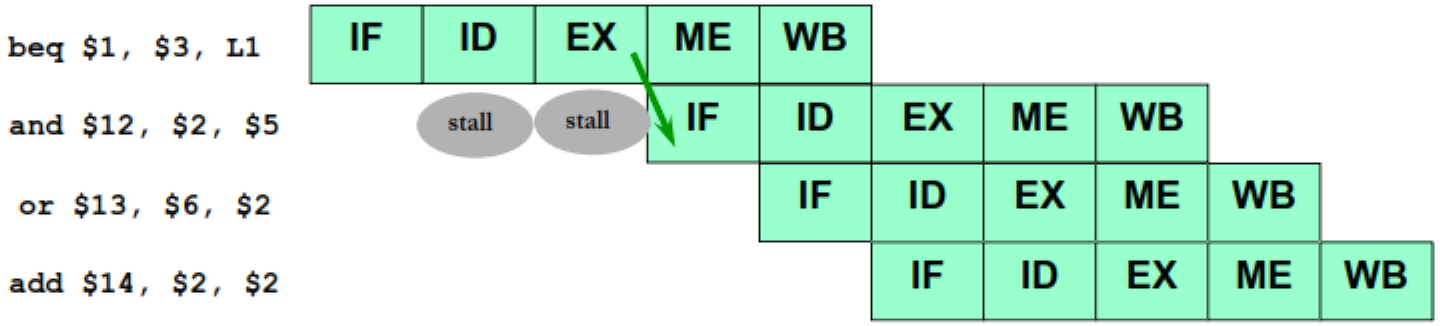
- **Conservative assumption: To stall the pipeline until the** branch decision is taken **(stalling until resolution) and** then fetch the correct instruction flow.
- Without forwarding: for **three clock cycles**
- With forwarding: for **two clock cycles**
- If the branch is *not taken, the three cycles penalty is not* justified ➔ throughput reduction.
- We can assume the ***branch not taken, and flush the*** next 3 instructions in the pipeline only if the branch will be taken. (We cannot assume the ***branch taken because*** we don't know the branch target address)

Branch Stalls without Forwarding



- Conservative assumption: Stalling until resolution at the end of the ME stage.
- Each branch costs **three stalls to fetch the correct** instruction flow: (PC+4) or Branch Target Address

Branch Stalls with Forwarding



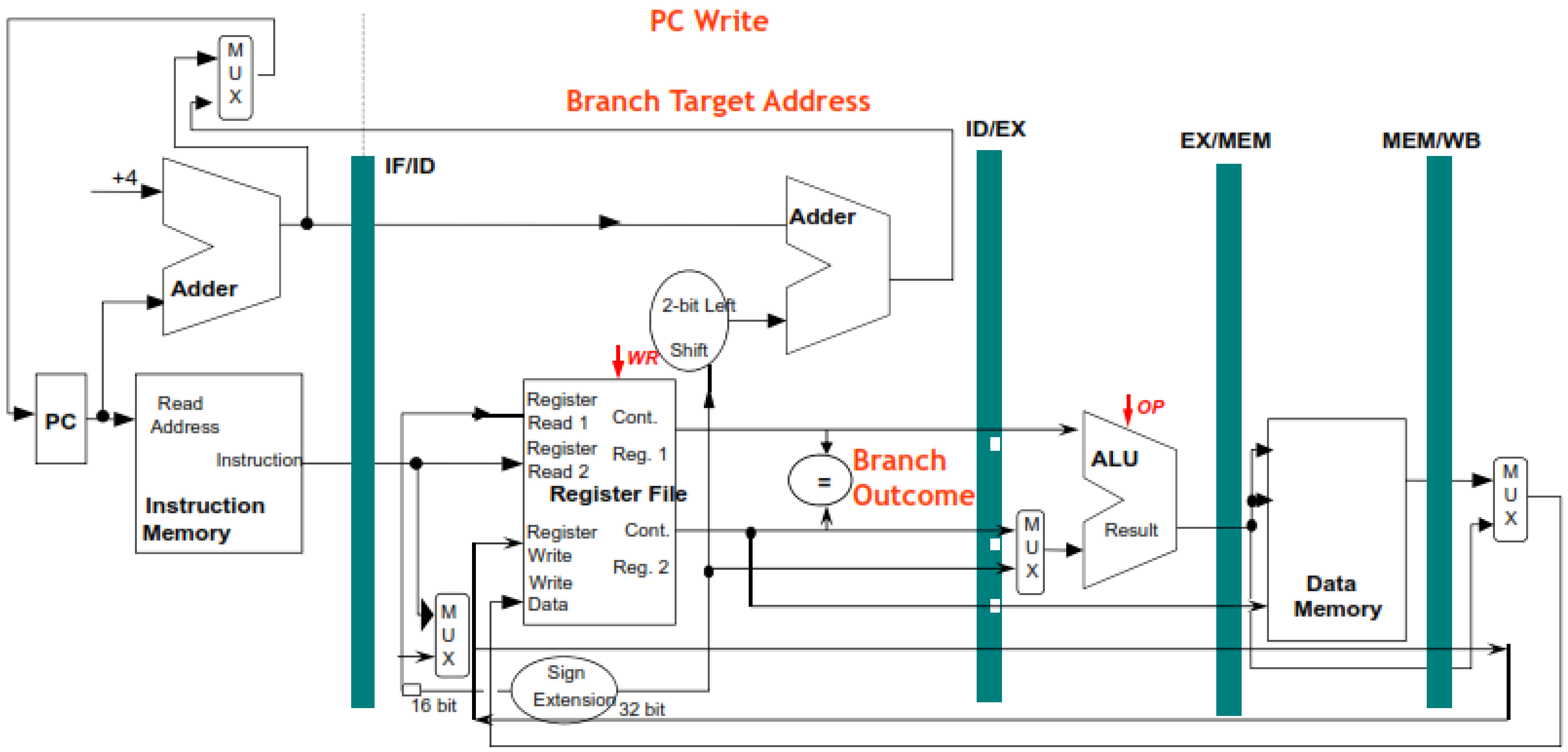
- Conservative assumption: Stalling until resolution at the end of the EX stage.
- Each branch costs **two stalls to fetch the correct** instruction flow: (PC+4) or Branch Target Address

Early Evaluation of the PC

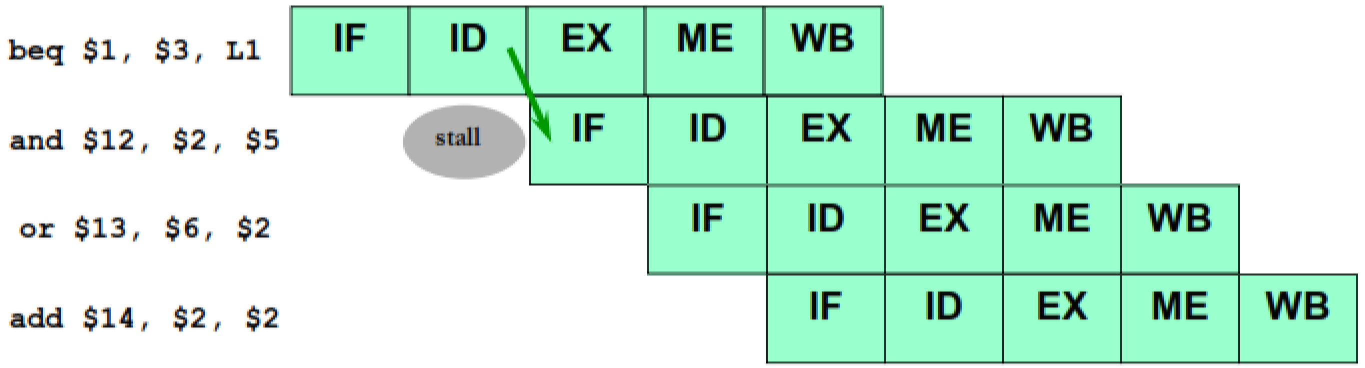
- To improve performance in case of branch hazards, we need to add hardware resources to:
  1. Compare registers to derive the **Branch Outcome**
  2. Compute the **Branch Target Address**
  3. Update the PC register

  as soon as possible in the pipeline.
- MIPS processor compares registers, computes branch target address and updates PC ***during ID stage.***

MIPS Processor: Early Evaluation of the PC



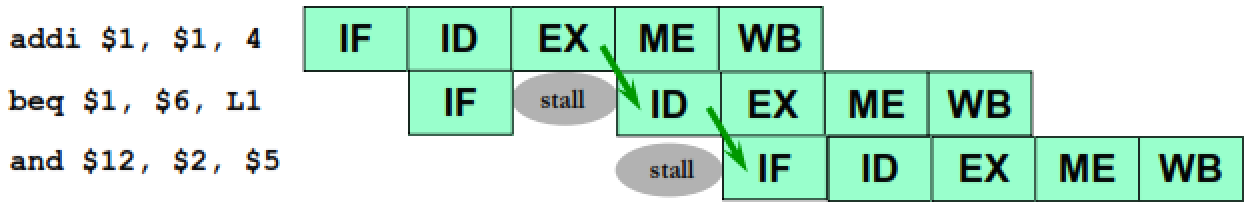
MIPS Processor: Early Evaluation of the PC



- Conservative assumption: Stalling until resolution at the end of the ID stage.
- Each branch costs **one stall to fetch the correct** instruction flow: (PC+4) or Branch Target Address

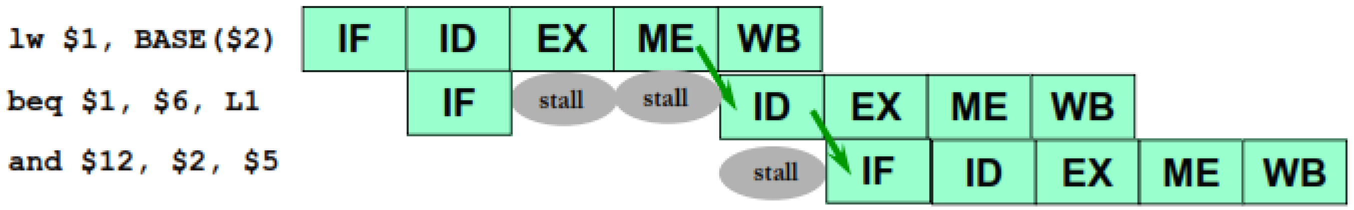
MIPS Processor: Early Evaluation of the PC

- Consequence of early evaluation of the branch decision in ID stage:
- In case of **add instruction followed by a branch** testing the result ➔ we need to introduce **one stall before ID stage of branch to enable the forwarding** (EX-ID) of the result from EX stage of previous instruction.
- As usual we need one stall **after the** branch for branch resolution



MIPS Processor: Early Evaluation of the PC

- Consequence of early evaluation of the branch decision in ID stage:
- In case of **load instruction followed by a branch** testing the result → we need to introduce **two stalls before ID stage of branch to enable the forwarding** (ME-ID) of the result from EX stage of previous instruction.
- As usual we need one stall **after the** branch for branch resolution.



MIPS Processor: Early Evaluation of the PC

- With the branch decision made during ID stage, there is a reduction of the cost associated with each branch **(branch penalty):**
  - We need only **one-clock-cycle stall after each branch**
  - Or a **flush of only one instruction following the** branch
- One-cycle-delay for every branch still yields a performance loss of 10% to 30% depending on the branch frequency:
- Pipeline Stall Cycles per Instruction due to Branches = Branch frequency x Branch Penalty
- We will examine some techniques to deal with this performance loss.

BRANCH PREDICTION TECHNIQUES

Branch Prediction Techniques

- In general, the problem of the branch becomes more important for deeply pipelined processors because the cost of incorrect predictions increases (the branches are solved several stages after the ID stage)
- **Main goal of branch prediction techniques: try to predict ASAP the** outcome of a branch instruction.
- The performance of a branch prediction technique depends on:
  - **Accuracy measured in terms of percentage of incorrect predictions given** by the predictor.
  - **Cost of a incorrect prediction measured in terms of time lost to execute** useless instructions **(misprediction penalty) given by the processor architecture**
- We also need to consider **branch frequency: the importance of** accurate branch prediction is higher in programs with higher branch frequency.

Branch Prediction Techniques

- There are many methods to deal with the performance loss due to branch hazards:
- **Static Branch Prediction Techniques: The actions for** a branch are fixed for each branch during the entire execution.
- The actions are fixed at compile time.
- **Dynamic Branch Prediction Techniques: The** decision causing the branch prediction can dynamically change during the program execution.
- In both cases, care must be taken not to change the processor state and registers until the branch is definitely known.

Static Branch Prediction Techniques

- Static Branch Prediction is used in processors where the expectation is that the branch behavior is highly predictable at compile time.
- Static Branch Prediction can also be used to assist dynamic predictors.

Static Branch Prediction Techniques

- 1) Branch Always Not Taken (Predicted-Not-Taken)
- 2) Branch Always Taken (Predicted-Taken)
- 3) Backward Taken Forward Not Taken (BTFNT)
- 4) Profile-Driven Prediction
- 5) Delayed Branch

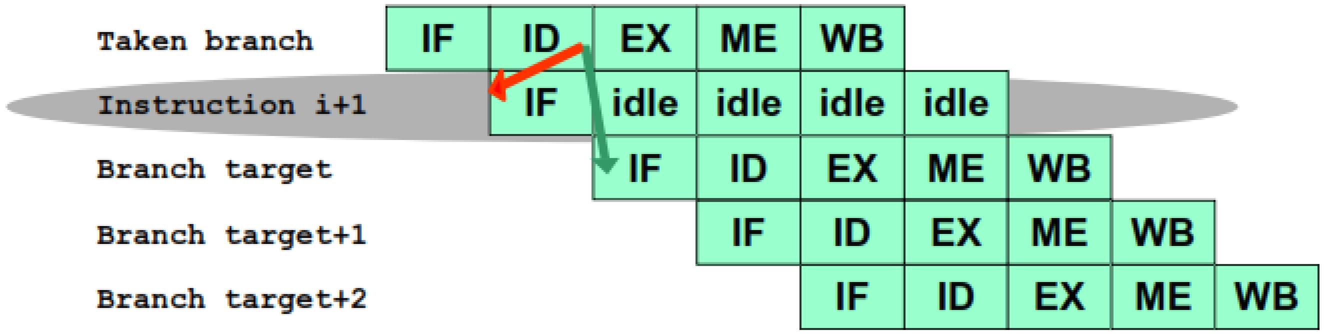
Branch Always Not Taken

- We assume the **branch will not be taken, thus the** sequential instruction flow we have fetched can continue as if the branch condition was not satisfied.
- If the condition in stage ID will result not satisfied **(the prediction is correct), we can preserve performance.**

Untaken branch	IF	ID	EX	ME	WB				
Instruction i+1		IF	ID	EX	ME	WB			
Instruction i+2			IF	ID	EX	ME	WB		
Instruction i+3				IF	ID	EX	ME	WB	
Instruction i+4					IF	ID	EX	ME	WB

Branch Always Not Taken

- If the condition in stage ID will result satisfied **(the prediction is incorrect), the branch is taken:**
- We need to **flush the next instruction already fetched (the** fetched instruction is turned into a **nop) and we restart the** execution by fetching the instruction at the branch target address ➔ **One-cycle penalty**



Taken branch	IF	ID	EX	ME	WB				
Instruction i+1		IF	idle	idle	idle	idle			
Branch target			IF	ID	EX	ME	WB		
Branch target+1				IF	ID	EX	ME	WB	
Branch target+2					IF	ID	EX	ME	WB

Branch Always Not Taken

- If on average the 50% of branches are not taken ➔ with this hypothesis we can halve the cost of control hazards
- If on average the probability that the branch is not taken is high ➔ with this hypothesis we can optimize the cost of control hazards.

Branch Always Taken

- An alternative scheme is to consider every branch as taken: as soon as the branch is decoded and the **Branch Target Address** is computed, we assume the branch to be taken and we begin fetching and executing at the target.
- The predicted-taken scheme makes sense for pipelines where the branch target address is known **before the branch** outcome.
- In MIPS pipeline, we don’t know the branch target address earlier than the branch outcome, so there is no advantage in this approach for this pipeline.
- We should anticipate the computation of BTA at the IF stage or we need a **Branch Target Buffer.**

Backward Taken Forward Not Taken (BTFNT)

- **The prediction is based on the branch direction**:
- Backward-going branches are predicted as taken
- **Example**: the branches at the end of loops go back at the beginning of the next loop iteration ➔ we assume the backward-going branches are always taken.
- Forward-going branches are predicted as not taken

Profile-Driven Prediction

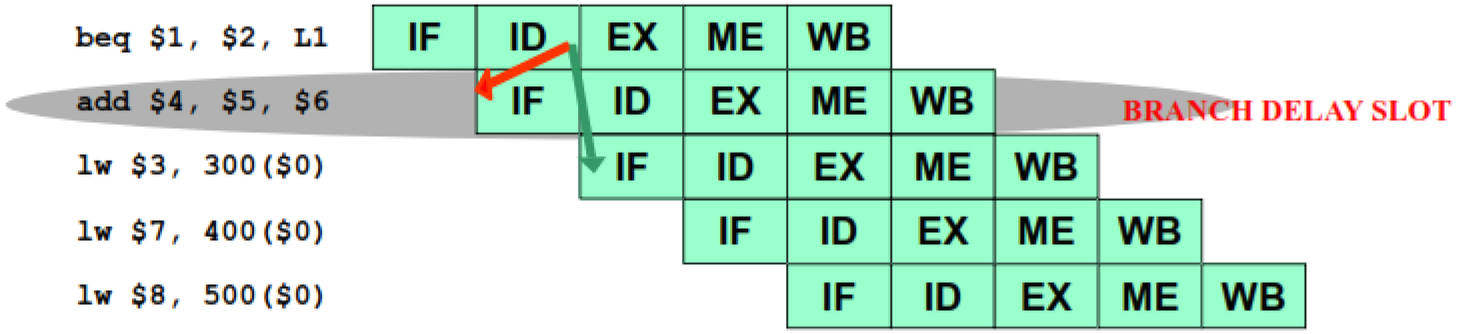
- The branch prediction is based on profiling information collected from earlier runs.
- The method can use compiler hints.

Delayed Branch Technique

- **Scheduling techniques**: The compiler statically schedules an independent instruction in the **branch delay slot.**
- The instruction in the branch delay slot is executed whether or not the branch is taken.
- If we assume a branch delay of one-cycle (as for MIPS) ➔ we have only **one-delay slot**
- Although it is possible to have for some deeply pipeline processors a branch delay longer than one-cycle ➔ almost all processors with delayed branch have a single delay slot (since it is usually difficult for the compiler to fill in more than one delay slot).

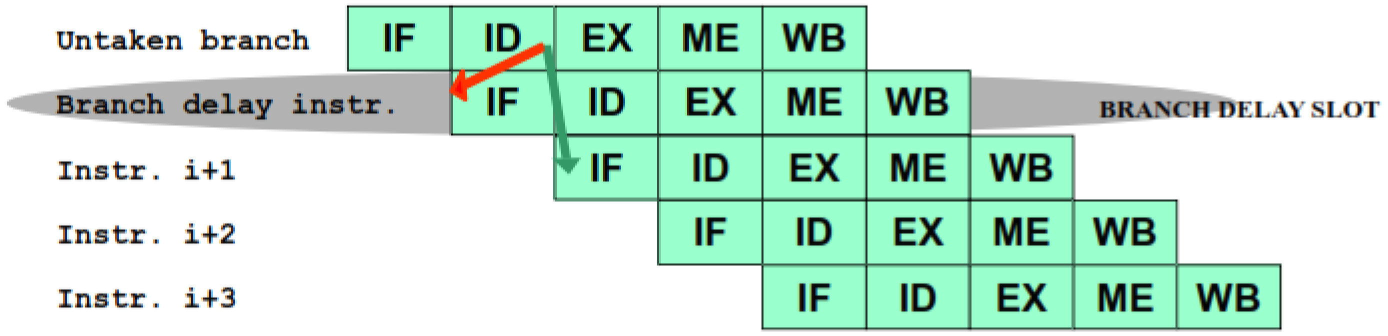
Delayed Branch Technique

- The MIPS compiler always schedules a branch independent instruction after the branch.
- **Example**: A previous **add instruction without any effects on** the branch is scheduled in the **Branch Delay Slot**



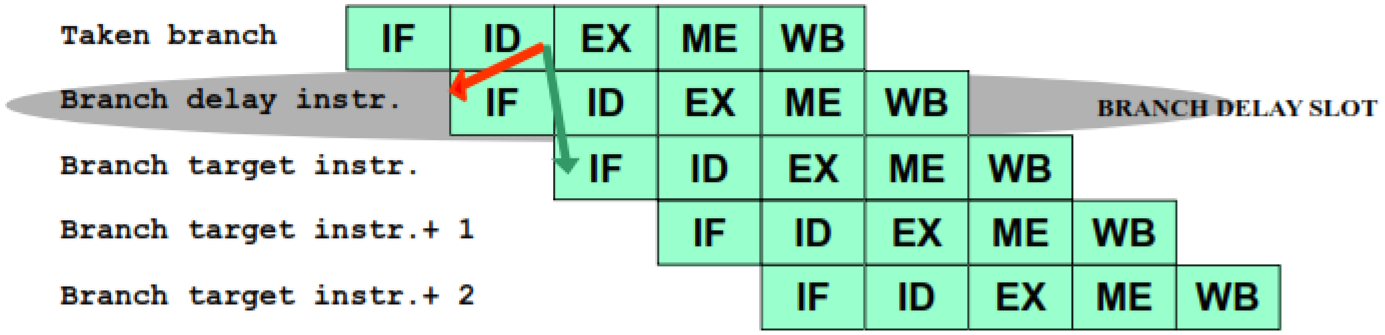
Delayed Branch Technique

- The behavior of the delayed branch is the same whether or not the branch is taken.
- If the branch is **untaken ➔ execution continues with** the instruction after branch



Delayed Branch Technique

- If the branch is **taken ➔ execution continues at the** branch target

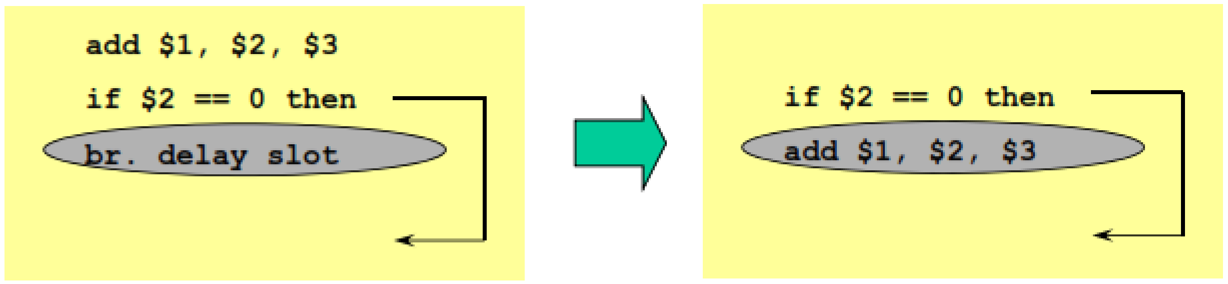


Delayed Branch Technique

- The job of the compiler is to make the instruction placed in the branch delay slot valid and useful.
- There are four ways in which the branch delay slot can be scheduled:
- **1. From before**
- **2. From target**
- **3. From fall-through**
- **4. From after**

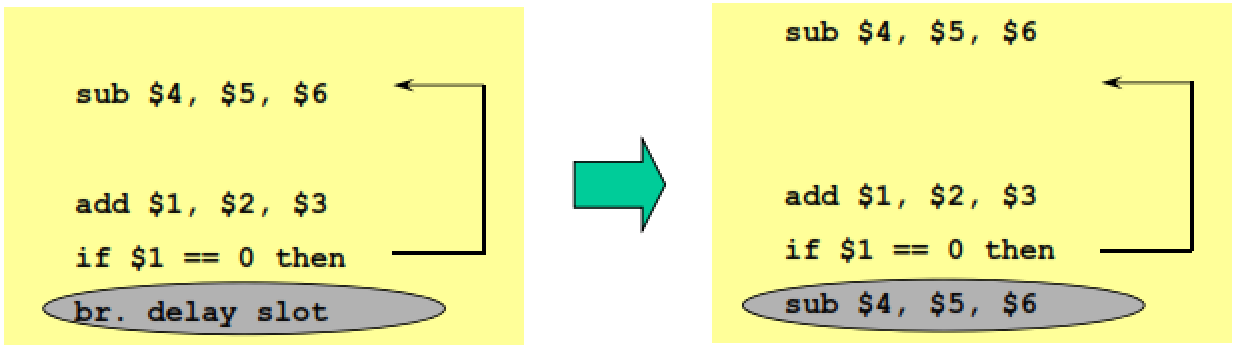
Delayed Branch Technique: From Before

- The branch delay slot is scheduled with an independent instruction from before the branch
- The instruction in the branch delay slot is **always executed (whether the branch is taken or untaken).**



Delayed Branch Technique: From Target

- The use of **$1 in the branch condition prevents add instruction** (whose destination is **$1) from being moved after the branch.**
- The branch delay slot is scheduled from the target of the branch (usually the target instruction will need to be copied because it can be reached by another path).
- This strategy is preferred when the branch is taken with high probability, such as loop branches **(backward branches).**

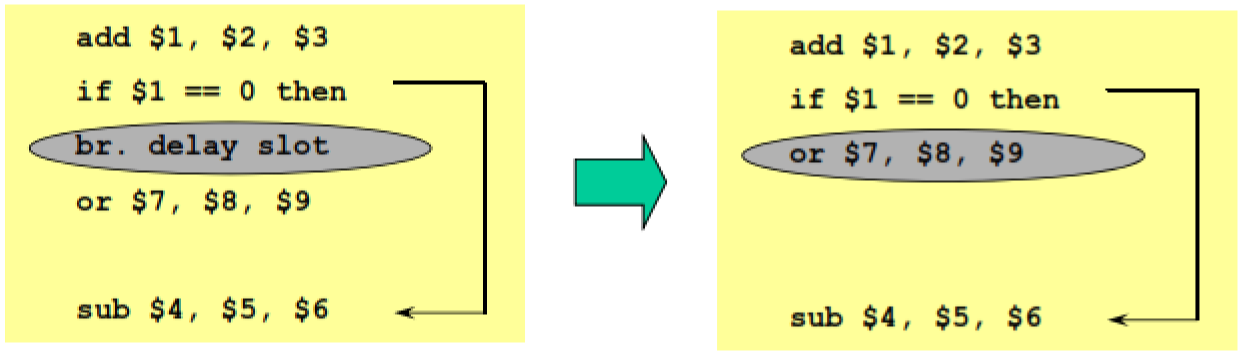


```
sub $4, $5, $6

add $1, $2, $3
if $1 == 0 then
br. delay slot
```

```
sub $4, $5, $6


add $1, $2, $3
if $1 == 0 then
sub $4, $5, $6
```

Delayed Branch Technique: From Fall-Through

- The use of $1 in the branch condition prevents add instruction (whose destination is $1) from being moved after the branch.
- The branch delay slot is scheduled from the not-taken fall-through path.
- This strategy is preferred when the branch is not taken with high probability, such as **forward branches.**



Delayed Branch Technique

- To make the optimization legal for the target an fall through cases, it must be OK to execute the moved instruction when the branch goes in the unexpected direction.
- By OK we mean that the instruction in the branch delay slot is executed but the work is wasted (the program will still execute correctly).
- For example, if the destination register is an unused temporary register when the branch goes in the unexpected direction.

Delayed Branch Technique

- In general, the compilers are able to fill about 50% of delayed branch slots with valid and useful instructions, the remaining slots are filled with **nops.**
- In deeply pipeline, the delayed branch is longer that one cycle: many slots must be filled for every branch, thus it is more difficult to fill all the slots with useful instructions.

Delayed Branch Technique

- The main limitations on delayed branch scheduling arise from:
- The restrictions on the instructions that can be scheduled in the delay slot.
- The ability of the compiler to statically predict the outcome of the branch.

Delayed Branch Technique

- To improve the ability of the compiler to fill the branch delay slot ➔ most processors have introduced a **canceling or nullifying branch: the instruction includes** the direction that the branch was predicted.
- When the branch behaves as predicted ➔ the instruction in the branch delay slot is executed normally.
- When the branch is incorrectly predicted ➔ the instruction in the branch delay slot is turned into a NOP
- In this way, the compiler need not be as conservative when filling the delay slot.

Delayed Branch Technique

- MIPS architecture has the **branch-likely instruction, that** behaves as cancel-if-not-taken branch:
  - The instruction in the branch delay slot is executed whether the branch is taken.
  - The instruction in the branch delay slot is **not executed (it is turned to NOP) whether the branch is untaken.**
- Useful approach for backward branches (such as loop branches).
- The branch delay slot must be filled from target.

DYNAMIC BRANCH PREDICTION TECHNIQUES

Dynamic Branch Prediction

- **Basic Idea: To use the past branch behavior to predict** the future.
- We use hardware to **dynamically predict the outcome of** a branch: the prediction will depend on the behavior of the branch at run time and will change if the branch changes its behavior during execution.
- We start with a simple branch prediction scheme and then examine approaches that increase the branch prediction accuracy.

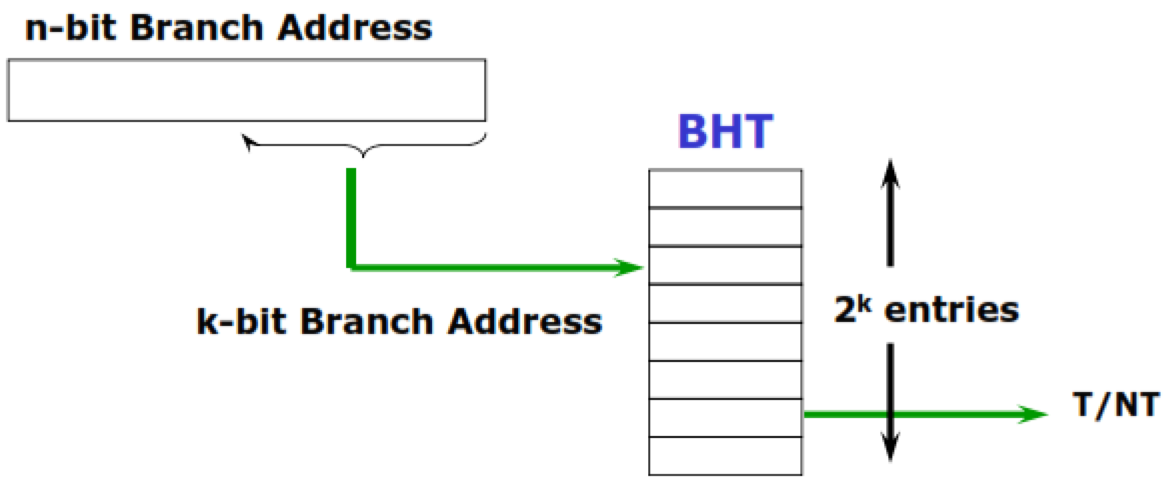
Dynamic Branch Prediction Schemes

- Dynamic branch prediction is based on two interacting mechanisms:
- **Branch Outcome Predictor:**
  - To predict the direction of a branch (i.e. taken or not taken).
- **Branch Target Predictor or Branch Target Buffer:**
  - To predict the branch target address in case of taken branch.
- These modules are used by the Instruction Fetch Unit to predict the next instruction to read in the I-cache.
  - If branch is not taken ➔ PC is incremented.
  - If branch is taken ➔ BTP gives the target address

Branch History Table

- **Branch History Table (or Branch Prediction Buffer)**:
    - Table containing 1 bit for each entry that says whether the branch was recently taken or not.
    - Table indexed by the lower portion of the address of the branch instruction.
- **Prediction**: hint that it is assumed to be correct, and fetching begins in the predicted direction.
    - If the hint turns out to be wrong, the prediction bit is inverted and stored back.
    - The pipeline is flushed and the correct sequence is executed.
- The table has no tags (every access is a hit) and the prediction bit could has been put there by another branch with the same low-order address bits: but it doesn’t matter. The prediction is just a hint!

Branch History Table



Accuracy of the Branch History Table

- A misprediction occurs when:
- The prediction is incorrect for that branch

or

- The same index has been referenced by two different branches, and the previous history refers to the other branch.
- To solve this problem it is enough to increase the number of rows in the BHT or to use a hashing function.

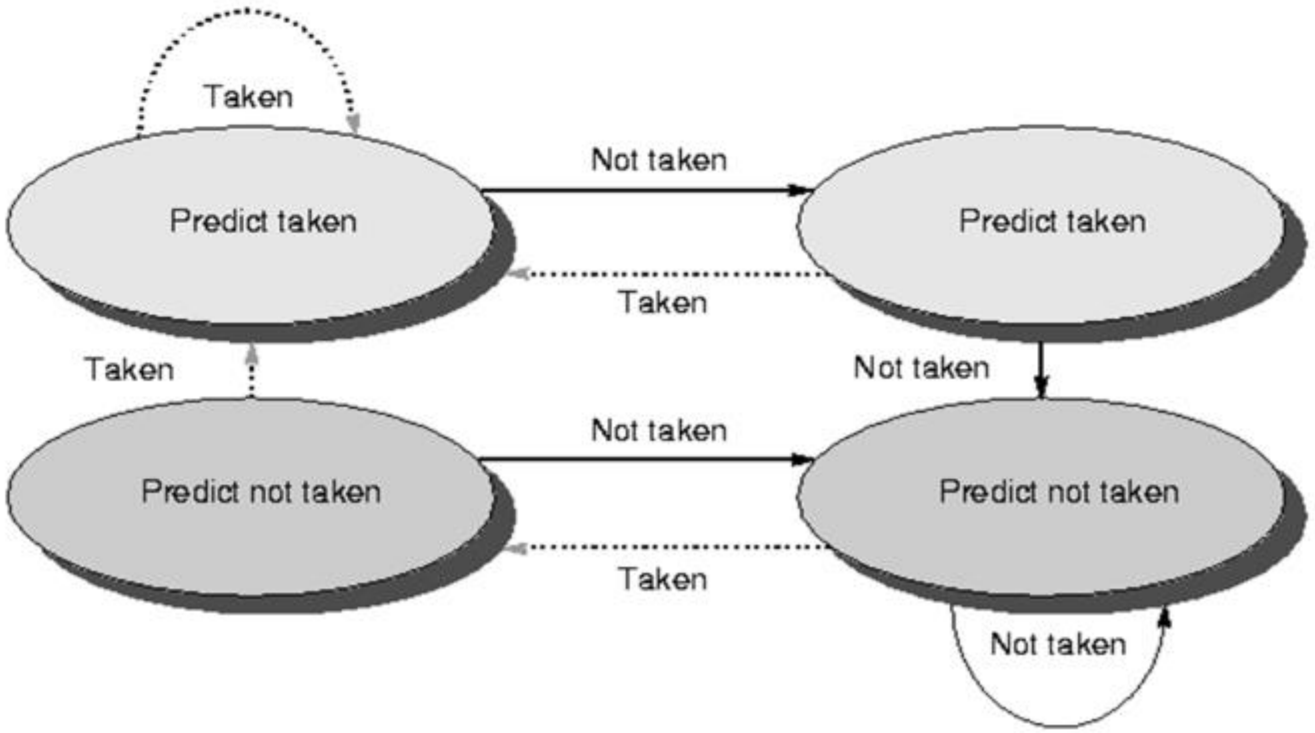
1-bit Branch History Table

- **Shortcoming of the 1-bit BHT:**
- In a loop branch, even if a branch is almost always taken and then not taken once, the 1-bit BHT will mispredict twice (rather than once) when it is not taken.
- **That scheme causes two wrong predictions:**
- At the last loop iteration, since the prediction bit will say taken, while we need to exit from the loop.
- When we re-enter the loop, at the end of the first loop iteration we need to take the branch to stay in the loop, while the prediction bit say to exit from the loop, since the prediction bit was flipped on previous execution of the last iteration of the loop.
- **For example**, if we consider a loop branch whose behavior is taken nine times and not taken once, the prediction accuracy is only 80% (due to 2 incorrect predictions and 8 correct ones).

2-bit Branch History Table

- The prediction must miss twice before it is changed.
- In a loop branch, at the last loop iteration, we do not need to change the prediction.
- For each index in the table, the 2 bits are used to encode the four states of a finite state machine.

FSM for 2-bit Branch History Table



n-bit Branch History Table

- **Generalization**: n-bit saturating counter for each entry in the prediction buffer.
  - The counter can take on values between 0 and 2^n-1
  - When the counter is greater than or equal to one-half of its maximum value (2^n-1), the branch is predicted as taken.
  - Otherwise, it is predicted as untaken.
- As in the 2-bit scheme, the counter is incremented on a taken branch and decremented on an untaken branch.
- Studies on n-bit predictors have shown that 2-bit predictors behave almost as well.

Accuracy of 2-bit Branch History Table

- For IBM Power architecture executing SPEC89 benchmarks, a 4K-entry BHT with 2-bit per entry results in:
  - Prediction accuracy from 99% to 82% (i.e. misprediction rate from 1% to 18%)
  - Almost similar performance with respect to an infinite buffer with 2-bit per entry.

Correlating Branch Predictors

- The 2-bit BHT uses only the recent behavior of a single branch to predict the future behavior of that branch.
- **Basic Idea**: the behavior of recent branches are correlated, that is the recent behavior of other branches rather than just the current branch (we are trying to predict) can influence the prediction of the current branch.

Hprcse

Example of Correlating Branches

```
    If(a==2)  a = 0; bb1
L1: If(b==2)  b = 0; bb2
L2: If(a!=b)  {};    bb3
```

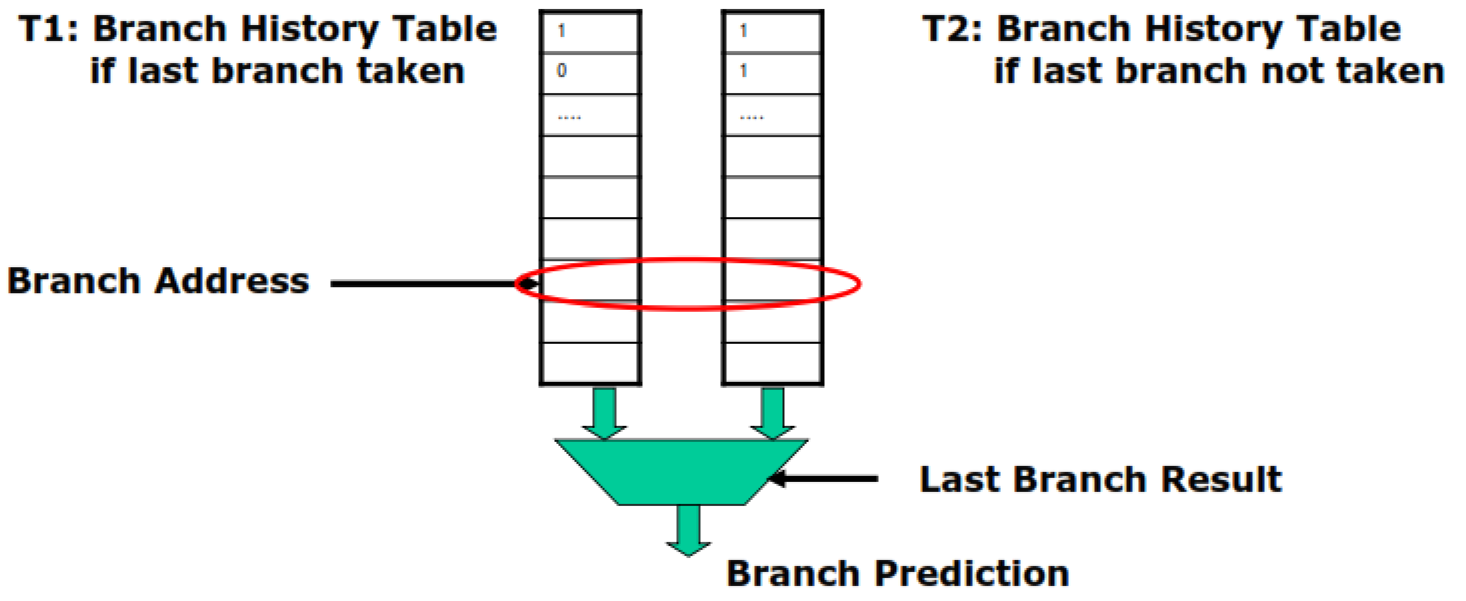
```
         subi r3,r1,2
         bnez r3,L1; bb1
         add  r1,r0,r0
L1:      subi r3,r2,2
         bnez r3,L2; bb2
         add  r2,r0,r0
L2:      sub  r3,r1,r2
         beqz r3,L3; bb3
L3:
```

- Branch bb3is correlated to previous branches bb1 and bb2.
- If previous branches are both not taken, then bb3 will be taken (a!=b)

Correlating Branch Predictors

- Branch predictors that use the behavior of other branches to make a prediction are called Correlating Predictors or 2-level Predictors.
- Example a (1,1) Correlating Predictors means a 1-bit predictor with 1-bit of correlation:
  - The behavior of last branch is used to choose among a pair of 1-bit branch predictors.

Correlating Branch Predictors: Example


T1: Branch History Table if last branch taken
1
0
.....
1
1
.....
T2: Branch History Table if last branch not taken
Branch Address
Last Branch Result
Branch Prediction

Correlating Branch Predictors

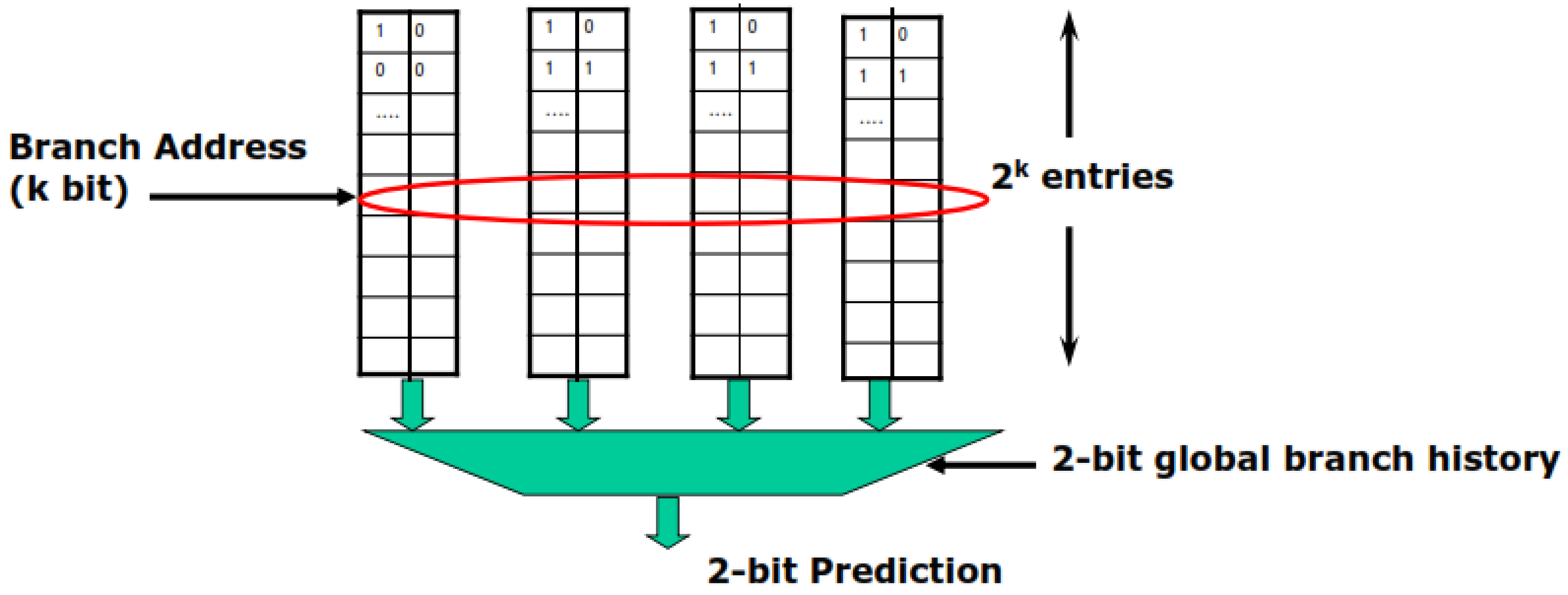
- Record if the k most recently executed branches have been taken or not taken.
- The branch is predicted based on the previous executed branch by selecting the appropriate 1-bit BHT:
  - One prediction is used if the last branch executed was taken
  - Another prediction is used if the last branch executed was not taken.
- In general, the last branch executed is not the same instruction as the branch being predicted (although this can occur in simple loops with no other branches in the loops).

(m, n) Correlating Branch Predictors

- In general (m, n) correlating predictor records last m branches to choose from 2m BHTs, each of which is a n-bit predictor.
- The branch prediction buffer can be indexed by using a concatenation of low-order bits from the branch address with m-bit global history (i.e. global history of the most recent m branches).

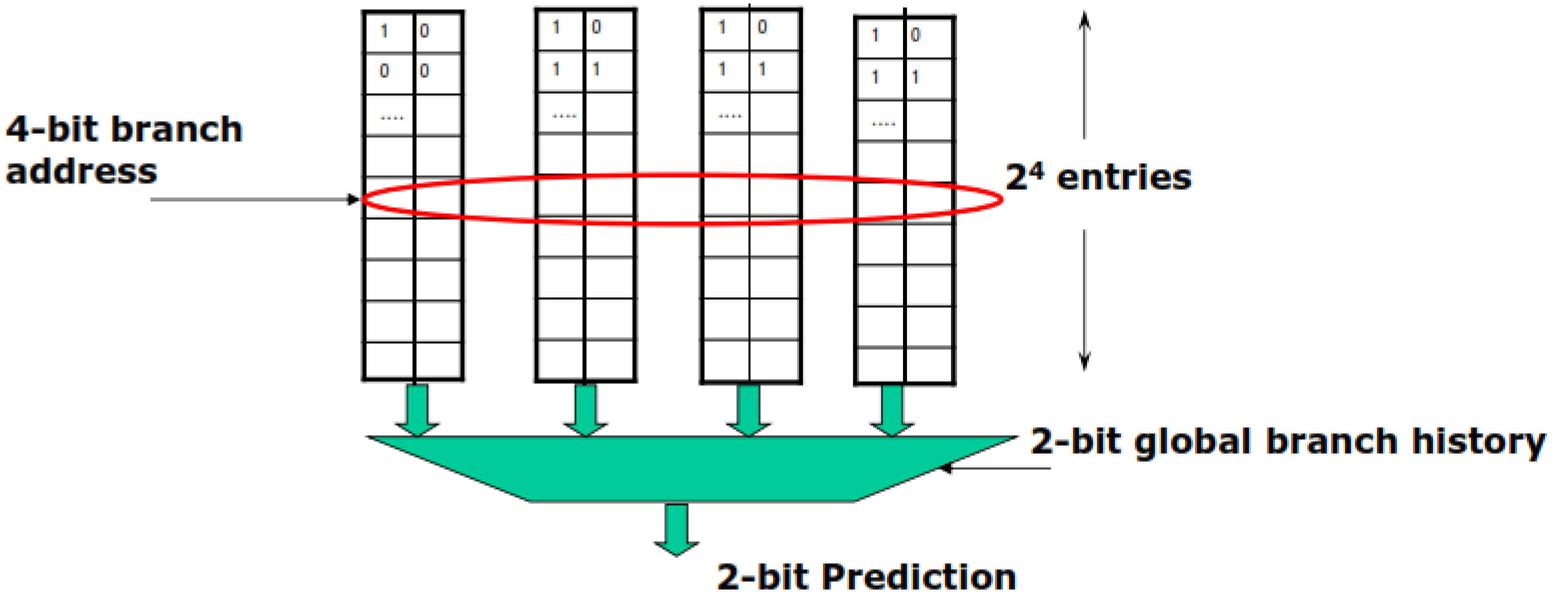
(2, 2) Correlating Branch Predictors

- A (2, 2) correlating predictor has 4 2-bit Branch History Tables.
- It uses the 2-bit global history to choose among the 4 BHTs.



Example of (2, 2) Correlating Predictor

- Example: a (2, 2)correlating predictor with 64 total entries ➔ 6-bit index composed of: 2-bit global history and 4-bit low-order branch address bits



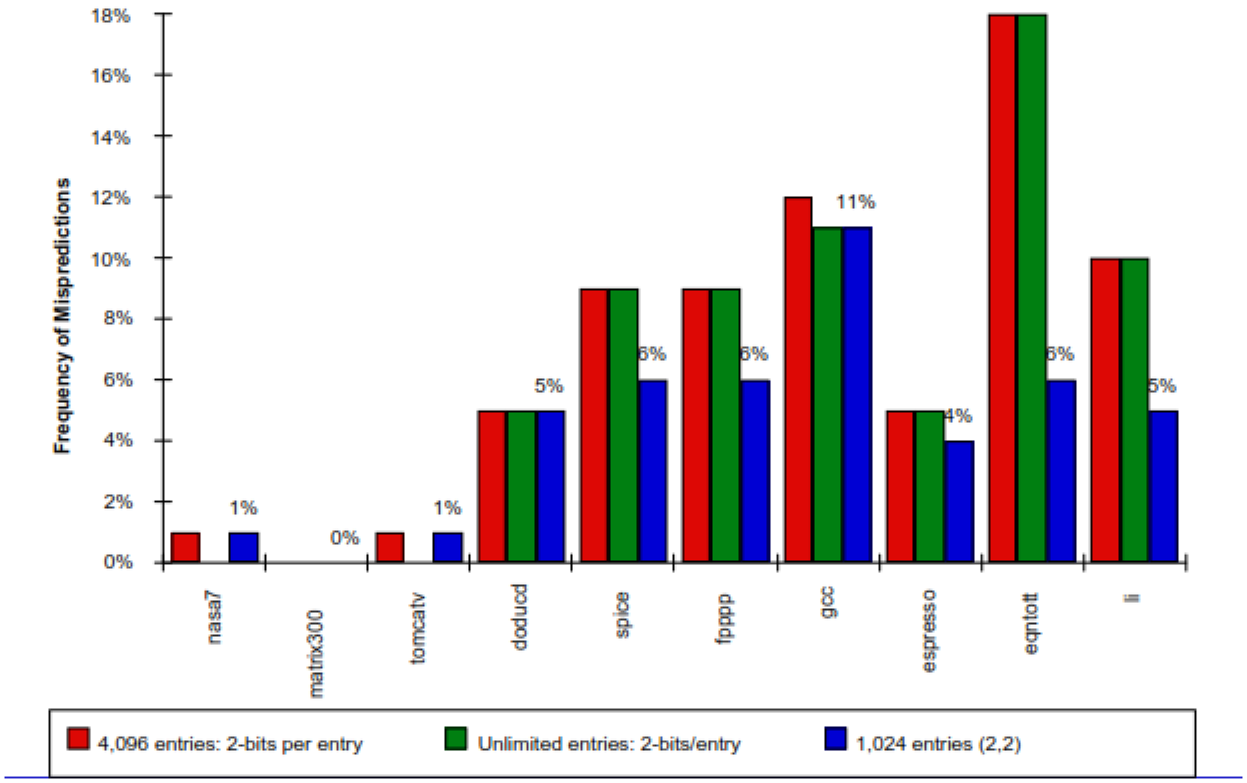
Example of (2, 2) Correlating Predictor

- Each BHT is composed of 16 entries of 2-bit each.
- The 4-bit branch address is used to choose four entries (a row).
- 2-bit global history is used to choose one of four entries in a row (one of four BHTs)

Accuracy of Correlating Predictors

- A 2-bit predictor with no global history is simply a (0, 2) predictor.
- By comparing the performance of a 2-bit simple predictor with 4K entries and a (2,2) correlating predictor with 1K entries.
- The (2,2) predictor not only outperforms the simply 2-bit predictor with the same number of total bits (4K total bits), it often outperforms a 2-bit predictor with an unlimited number of entries.

Accuracy of Correlating Predictors

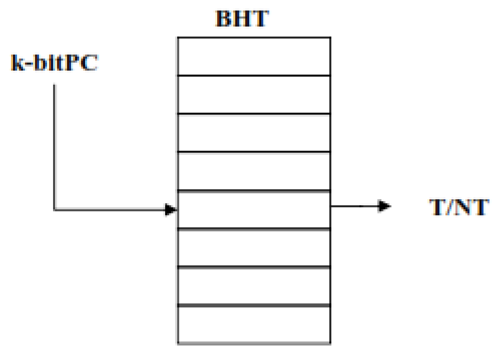


Two-Level Adaptive Branch Predictors

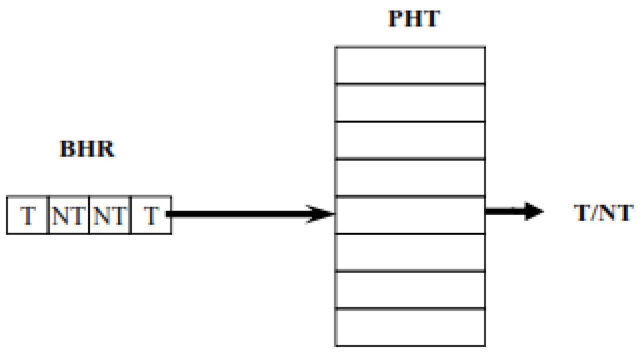
- The first level history is recorded in one (or more) k-bit shift register called Branch History Register (BHR), which records the outcomes of the k most recent branches
- The second level history is recorded in one (or more) tables called Pattern History Table (PHT)of two-bit saturating counters
- The BHR is used to index the PHT to select which 2-bit counter to use.
- Once the two-bit counter is selected, the prediction is made using the same method as in the two-bit counter scheme.

GA Predictor

- BHT: Local predictor
- Indexed by the low order bits of PC (branch address)

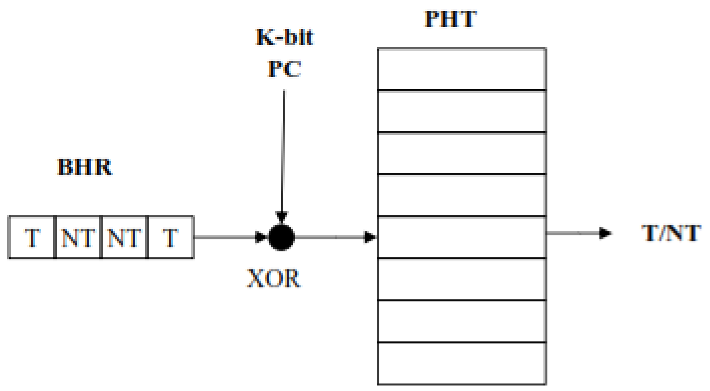


- GAs: Local and global predictor
- 2-level predictor: PHT Indexed by the content of BHR (global history)



GShare Predictor

- GShare: Local XOR global information
- Indexed by the exclusive OR of the low-order bits of PC (branch address) and the content of BHR (global history)



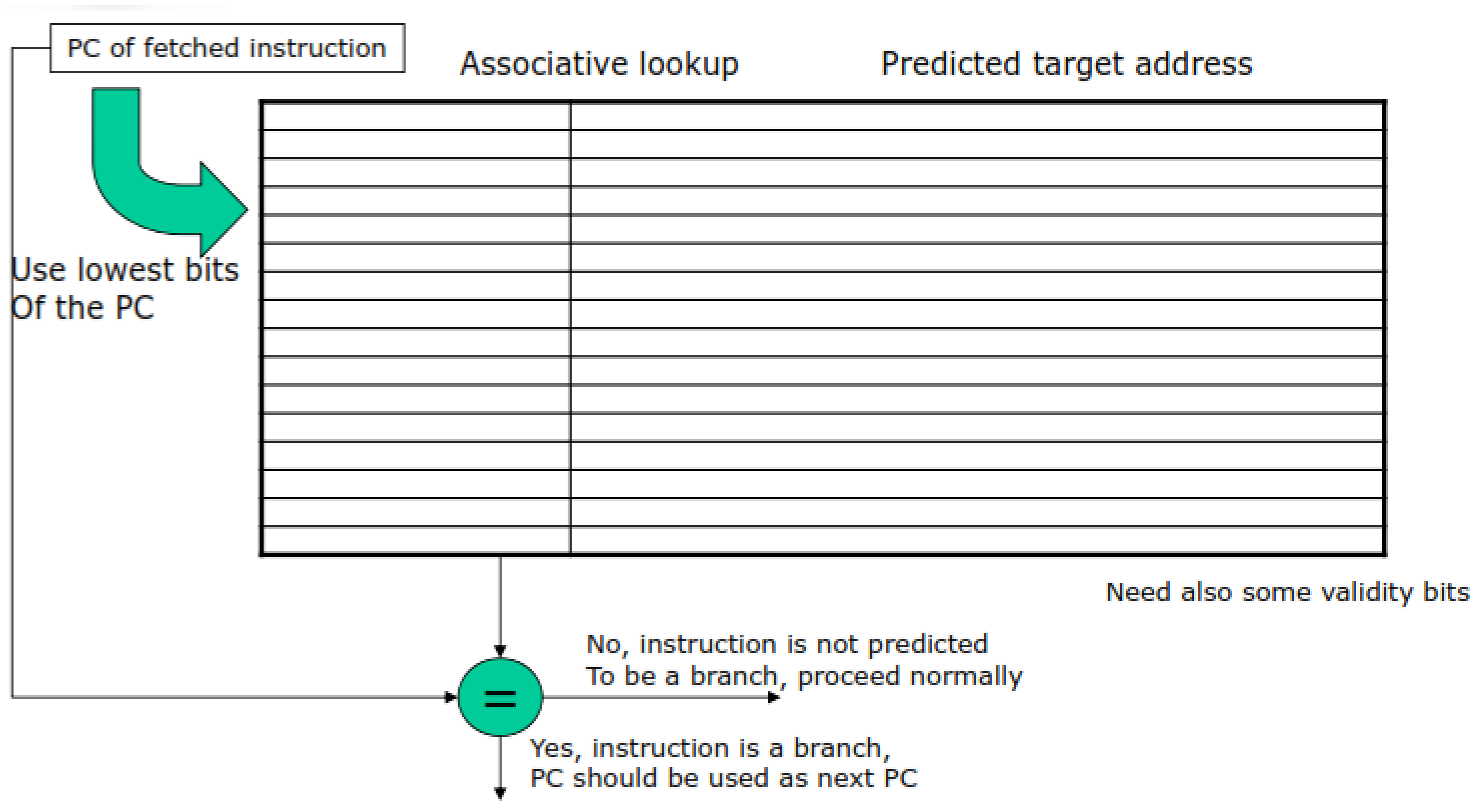
Branch Target Buffer

- **Branch Target Buffer (Branch Target Predictor)** is a cache storing the predicted branch target address for the next instruction after a branch
- We access the BTB in the IF stage using the instruction address of the fetched instruction (a possible branch) to index the cache.
- Typical entry of the BTB:

Exact Address of a Branch	Predicted target address

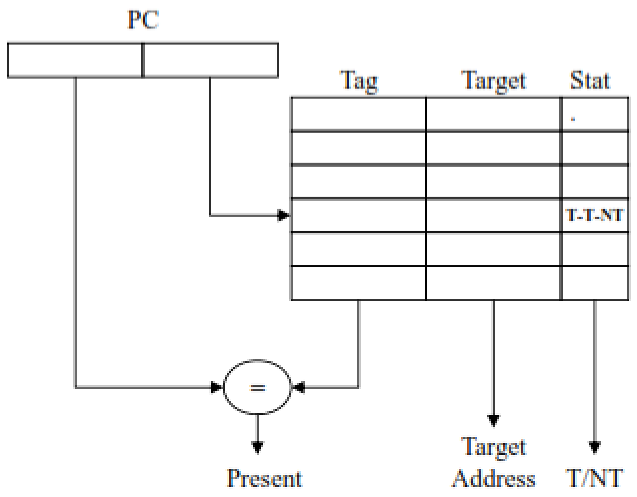
- The predicted target address is expressed as PC-relative

Structure of a Branch Target Buffer


PC of fetched instruction
Associative lookup
Predicted target address
Use lowest bits
Of the PC
Need also some validity bits
=
No, instruction is not predicted
To be a branch, proceed normally
Yes, instruction is a branch,
PC should be used as next PC

Structure of a Branch Target Buffer

- In the BTB we need to store the predicted target address only for taken branches.
- BTB entry:
- Tag + Predicted target address (expressed as PC relative for conditional branches) + Prediction state bits as in a Branch Outcome Predictor (optional).



Speculation

- Without branch prediction, the amount of parallelism is quite limited, since it is limited to within a basic block–a straight-line code sequence with no branches in except to the entry and no branches out except at the exit.
- Branch prediction techniques can help to achieve significant amount of parallelism.
- We can further exploit ILP across multiple basic blocks overcoming control dependences by speculating on the outcome of branches and executing instructions as if our guesses were correct.
- With speculation, we fetch, issue and execute instructions as if out branch predictions were always correct, providing a mechanism to handle the situation where the speculation is incorrect.
- Speculation can be supported by the compiler or by the hardware.

Hardware-Based Speculation

- Hardware-based speculation extends the ideas of dynamic scheduling and combines three keys ideas:
- Dynamic Branch Prediction to choose which instruction to execute;
- Speculation to execute instructions before control dependences are solved (with the ability to undo the effects of an incorrectly speculated sequence);
- Dynamic Scheduling to deal with the scheduling of different combinations of basic blocks.
- Hardware-based speculation follows the predicted flow of data values to choose when to execute instructions: operations execute as soon as their operands are available (data flow execution).

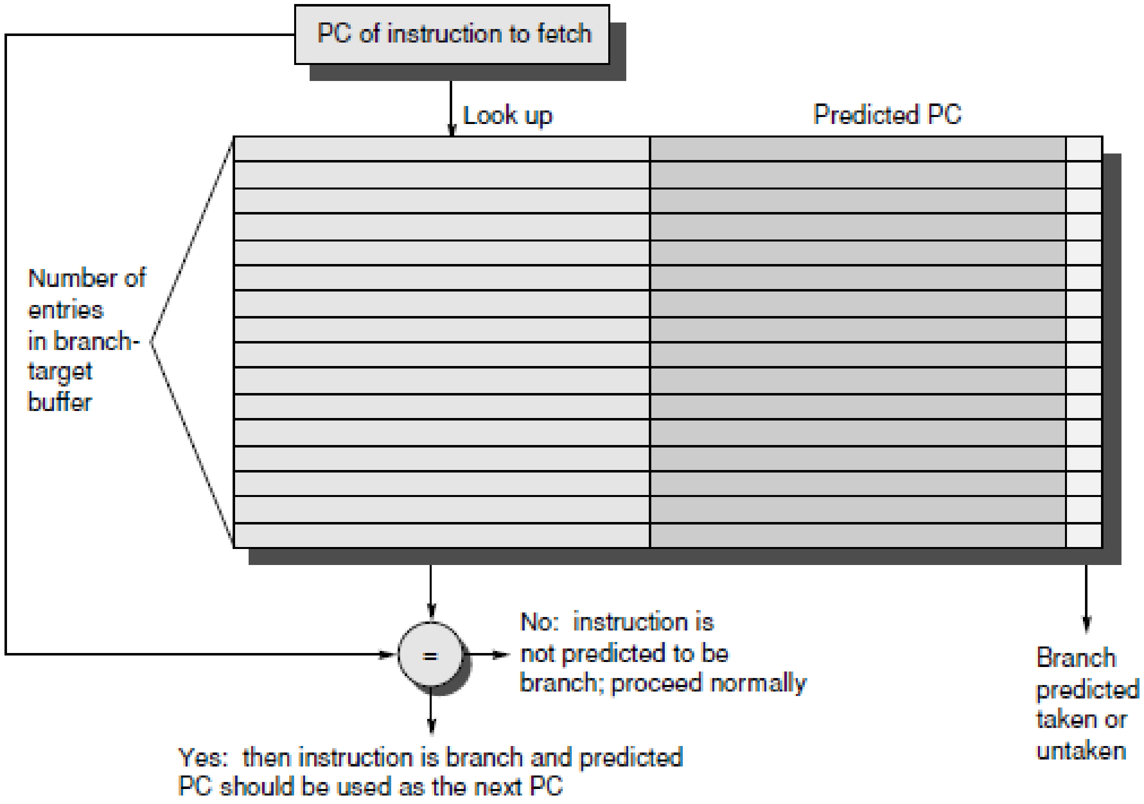
Hardware-Based Speculation

- When an instruction is no longer speculative, we allow it to update the register file or memory (instruction commit).
- The key idea behind speculation is to allow instructions to execute out-of –order but to force them to commit in-order and to prevent any irrevocable action (such as updating state or taking an exception) until an instructions commits.
- Reorder buffer (ROB) to hold the results of instructions that have completed execution but have not committed or to pass results among instructions that may be speculated.

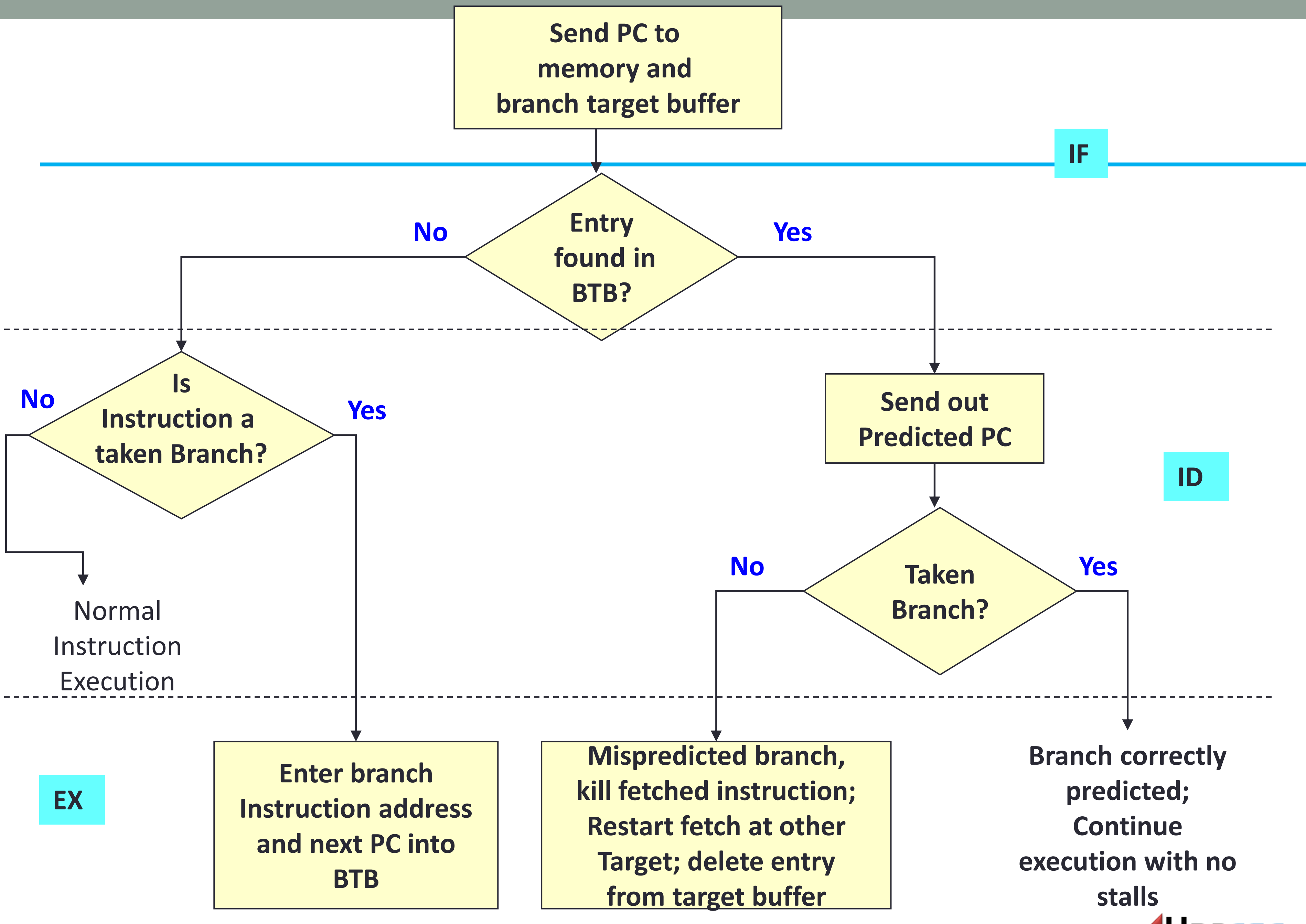
High Performance Instruction Delivery - Branch Target Buffer

- Branch Target Buffer (BTB): Address of branch index to get prediction AND branch address (if taken)
  - Note: must check for branch match now, since can't use wrong branch address
- Example:

BTB combined with BHT



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


Send PC to memory and branch target buffer
IF
Entry found in BTB?
No
Yes
Is Instruction a taken Branch?
No
Yes
Send out Predicted PC
ID
Normal Instruction Execution
Taken Branch?
No
Yes
EX
Enter branch Instruction address and next PC into BTB
Mispredicted branch, kill fetched instruction; Restart fetch at other Target; delete entry from target buffer
Branch correctly predicted; Continue execution with no stalls

High Performance Instruction Delivery - Branch Target Buffers

- Branch Target Buffers
  - Stores the predicted PC after a branch (Jump)
    - If a matching entry is found in the BTB, fetching begins immediately at the predicted PC.
- Target instruction buffers
  - Store one or more target instructions
    - Faster
    - Allows larger BTB
  - Allows branch folding
    - To obtain 0-cycle unconditional branches
    - substitute the instruction from the branch target buffer in place of the instruction that is returned from the cache

Hprcse

High Performance Instruction Delivery

- Integrated Instruction Fetch Units
  - A separate autonomous unit that feeds instructions to the rest of the pipeline.
  - Integrates several functions:
    - Integrated branch prediction
    - Instruction pre-fetch
    - Instruction buffering
- Return Address Predictors
  - A technique for predicting indirect jumps
    - Destination address varies at runtime
  - A small buffer of return addresses operating as a stack
    - caches the most recent return addresses

Hardware-Based Speculation

- Problem
  - A wide issue processor may need to execute a branch every clock cycle to maintain maximum performance.
    - Just predicting branches accurately may not be sufficient to generate the desired amount of ILP
- Solution
  - speculating on the outcome of branches and executing the program as if our guesses were correct
    - Fetch, issue, and execute instructions, as if branch predictions were always correct
  - Provide mechanisms to handle the situation where the speculation is incorrect.

Hardware-Based Speculation

- 3 key ideas
  - Dynamic branch prediction
    - to choose which instructions to execute
  - Speculation
    - To allow the execution of instructions before the control dependences are resolved
    - With the ability to undo the effects of an incorrectly speculated sequence
  - Dynamic scheduling
    - To deal with the scheduling of different combinations of basic blocks.

Hardware-Based Speculation

- Tomasulo’s algorithm can be extended to support speculation
  - separate the bypassing of results among instructions from the actual completion of an instruction
  - Allow an instruction to execute and to bypass its results to other instructions
    - without allowing the instruction to perform any updates that cannot be undone
    - Instructions using speculated results become speculative
  - When an instruction is no longer speculative, we allow it to update the register file or memory
    - instruction commit

Hardware-Based Speculation

- The key idea
  - Allow instructions to execute out of order but to force them to commit in order
  - Prevent any irrevocable action until an instruction commits
    - Such as updating state or taking an exception
  - Separate the process of completing execution from instruction commit
    - An additional set of hardware buffers that hold the results of instructions before being committed
  - reorder buffer (ROB)
    - A source of operands for instructions
    - Supplies operands in the interval between completion of instruction execution and instruction commit.

Hardware-Based Speculation

- ROB
  - A circular buffer
    - Entries allocated and de-allocated by two revolving pointers
  - Entries allocated to each instruction
    - Strictly in program order
    - Keeps track of the execution status of the instruction

Hardware-Based Speculation

- ROB fields
  - instruction type
    - opcode
      - branch (has no destination result)
      - store (has a memory address destination)
      - register operation (has register destinations).
  - destination
    - the register number or the memory address
  - value
    - Instruction result
  - Ready
    - The value is ready
  - Address
    - For load/store operation
- ROB replaces the store buffer

ROB[ ].Instruction
ROB[ ].Dest
ROB[ ].Value
ROB[ ].Ready
ROB[ ].A

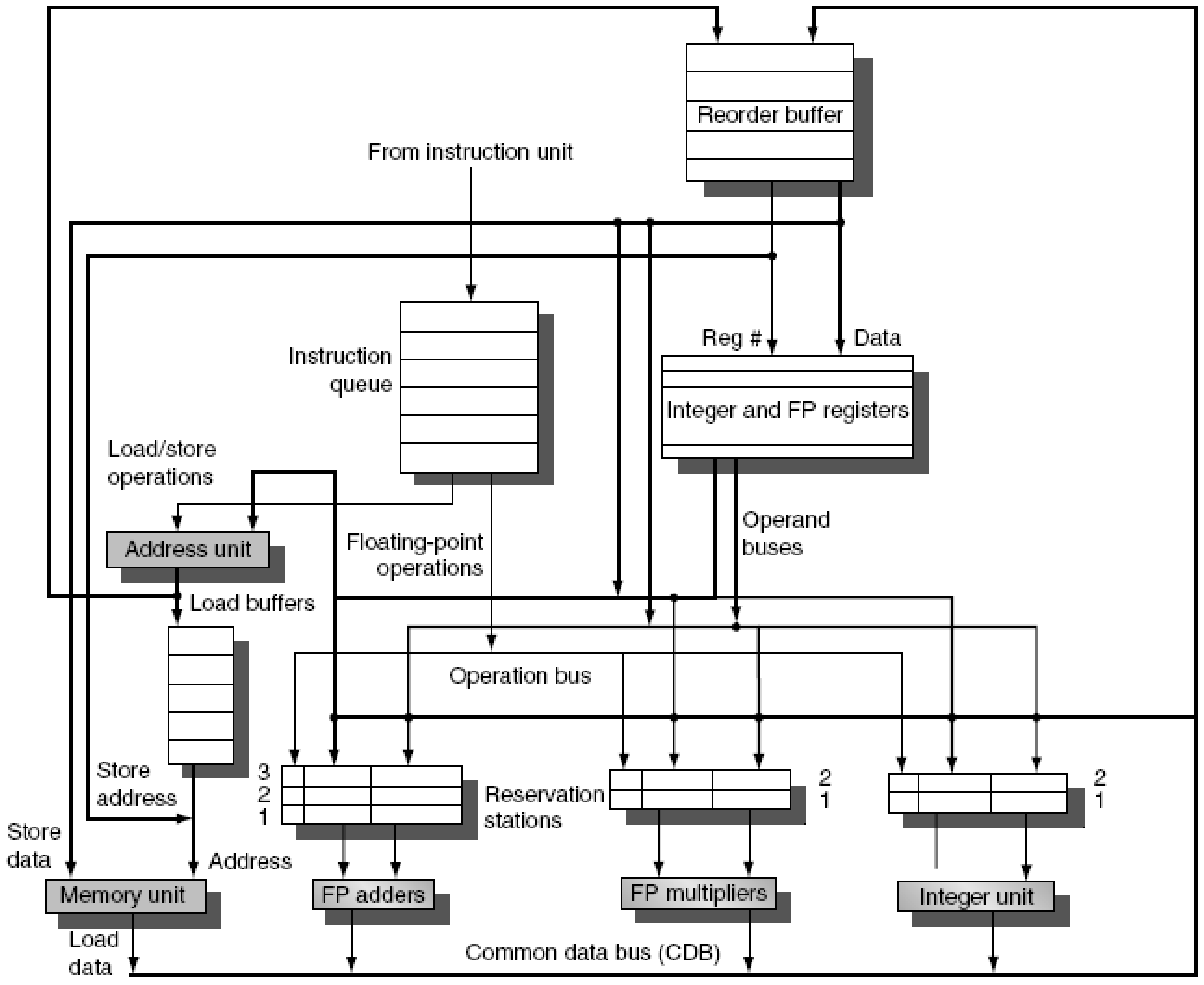
Type	**dest**	**value**	**Ready**

Hardware-Based Speculation

- Register fields
  - Busy
    - RegisterState[ ].Busy
  - Reorder
    - RegisterState[ ].Reorder
    - Instruction sequence number
  - Qi
    - RegisterState[ ].Qi
  - Value
    - RegisterState[ ].Value

Busy
reorder
Qi
Value

Hardware-Based Speculation

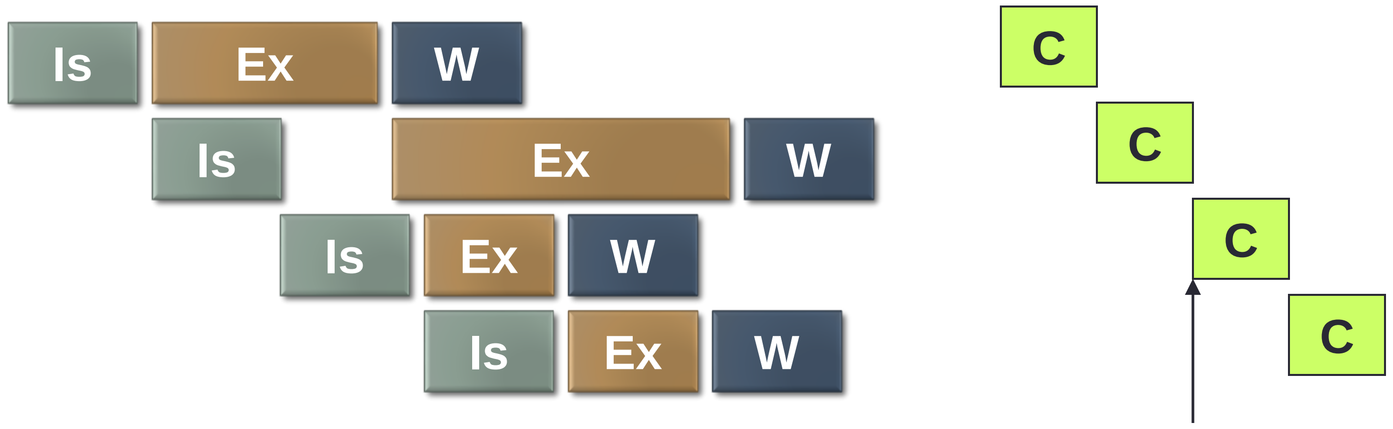


Hardware-Based Speculation – 4 Steps

- Issue
  - If there is an empty reservation station and empty slot in the ROB
    - Mark the reservation station and ROB as busy
  - Send the operands to the reservation station if they are available in the register or ROB
- Execute
- Write result
  - Write result on the CDB and from CDB into the ROB
  - Mark the reservation station as empty
- Commit
  - When an instruction reaches the head of the ROB
  - Mark the ROB as empty
  - When the instruction is a branch with incorrect prediction, indicate the speculation was wrong
    - The ROB is flushed
    - Execution is restarted at the correct successor of the branch

Hardware-Based Speculation

- Advantages of speculation
- Precise interrupt
  - The processor with the ROB can dynamically execute code while maintaining a precise interrupt model.
    - Flushing any pending instructions in ROB



Hardware-Based Speculation

- Advantages of speculation
  - Early recovery from branch misprediction
    - The processor can easily undo its speculative actions when a branch is found to be mispredicted.
      - Clearing the ROB for all entries that appear after the mispredicted branch
      - Allowing those that are before the branch in the ROB to continue
    - Performance is more sensitive to the branch prediction mechanism prefetch Pre-execution commit

Hardware-Based Speculation

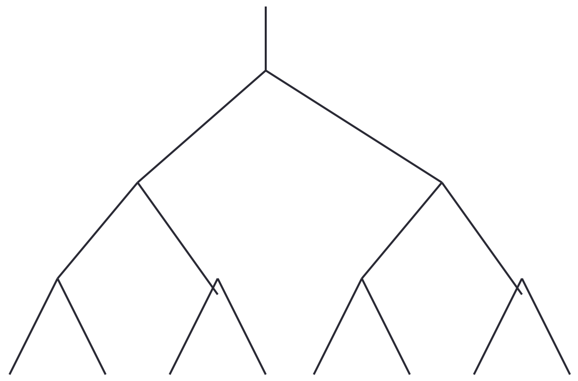
- Exception processing
  - If a speculated instruction raises an exception
    - the exception is recorded in the ROB.
    - not recognizing the exception until it is ready to commit.
    - the exception is flushed along with the instruction when the ROB is cleared
  - If instruction reaches the head of the ROB, it is no longer speculative
- Control complexity
  - Speculation adds significant complications to the control logic

Hardware-Based Speculation - Advantages of speculation

- Load and store hazard
  - A store updates memory only when it reaches the head of the ROB
  - WAW and WAR hazards through memory are eliminated with speculation
    - actual updating of memory occurs in order
  - RAW hazards through memory are maintained by
    - not allowing a load to initiate the second step of its execution
    - Check if any store has an Destination field that matches the value of the load
      - store r1, 100(r2)
      - load r3, 100(r2)

Hardware-Based Speculation

- Multiple Issue with Speculation
  - Process multiple instruction per clock cycle
  - Commit multiple instruction per clock cycle
- Challenges
  - Instruction issue
  - Monitoring CDBs for instruction completion



Hardware-Based Speculation

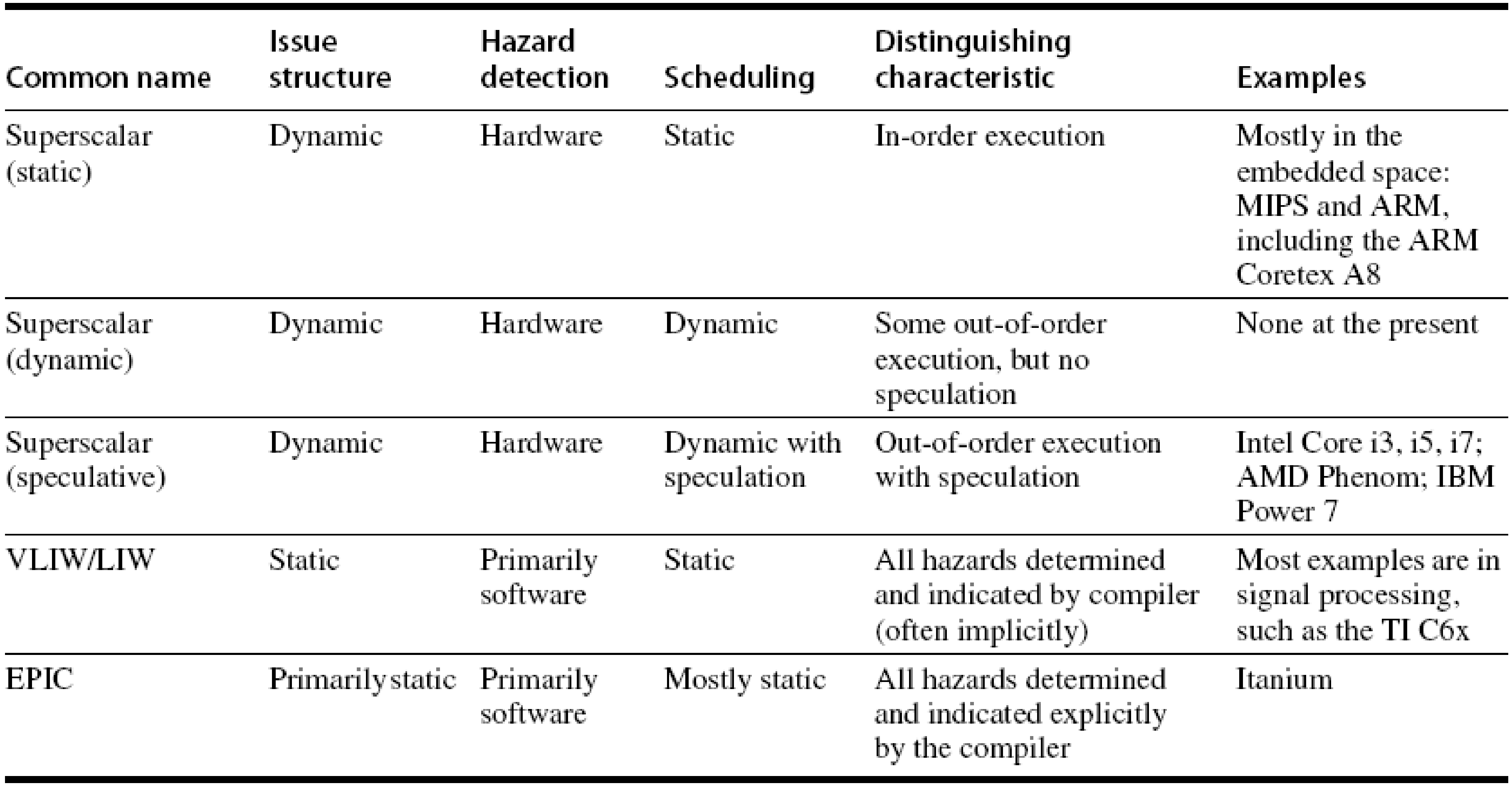
- Design Considerations for Speculative Machines
  - Register renaming versus reorder buffers
    - With speculation, register values may also temporarily reside in the ROB.
    - In register renaming approach, an extended registers is used to hold values.
  - How much to speculate
    - The cost of speculation is exceptional event
      - cache miss, TLB miss
  - Speculating through multiple branches
    - complicates the process of speculation recovery

Multiple Issue and Static Scheduling

- To achieve CPI < 1, need to complete multiple instructions per clock

- Solutions:
  - Statically scheduled superscalar processors
  - VLIW (very long instruction word) processors
  - dynamically scheduled superscalar processors

Multiple Issue



Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the ARM Coretex A8
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

VLIW Processors

- Package multiple operations into one instruction

- Example VLIW processor:
  - One integer instruction (or branch)
  - Two independent floating-point operations
  - Two independent memory references

- Must be enough parallelism in code to fill the available slots

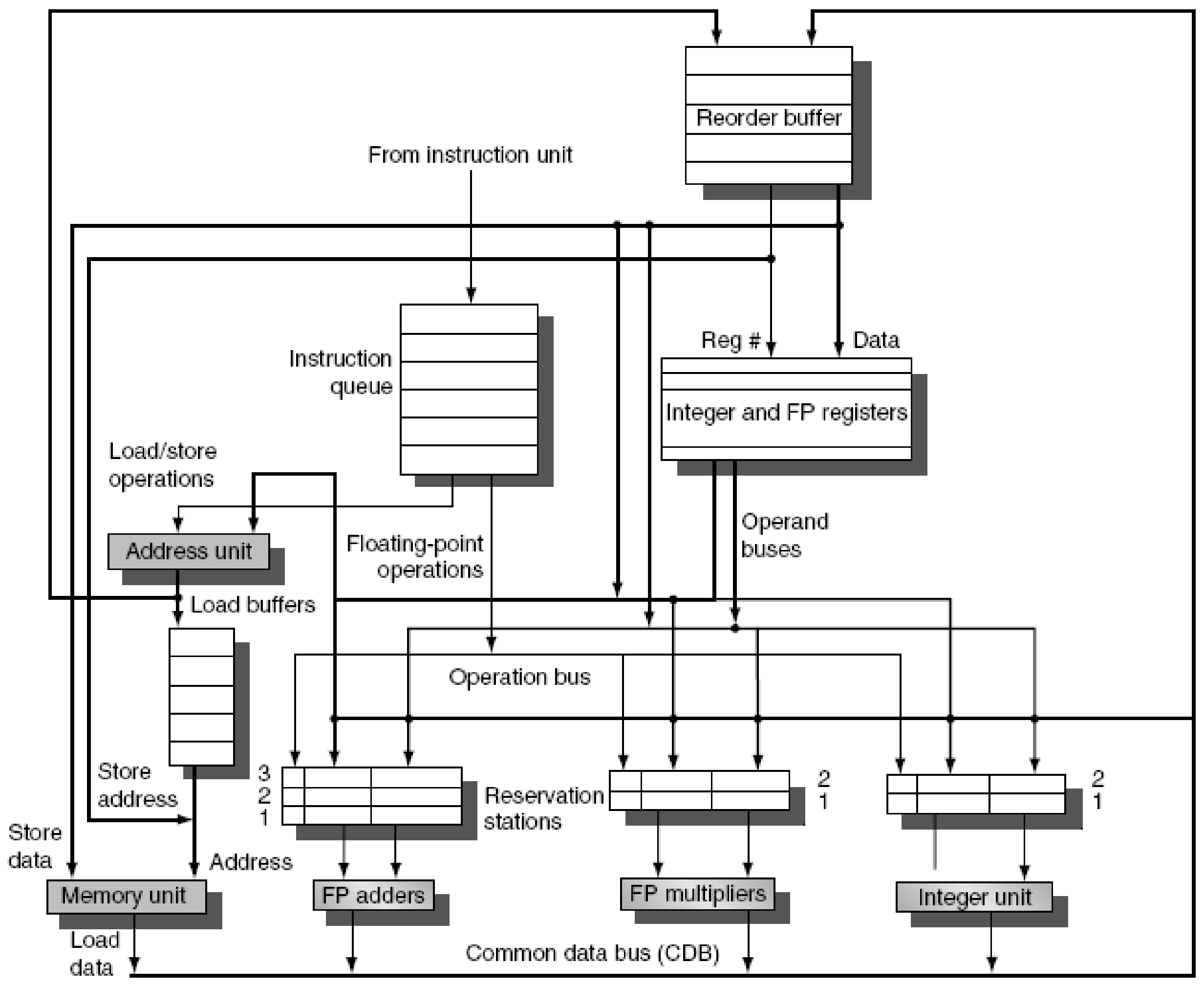
VLIW Processors

- Disadvantages:
  - Statically finding parallelism
  - Code size
  - No hazard detection hardware
  - Binary code compatibility

Dynamic Scheduling, Multiple Issue, and Speculation

- Modern microarchitectures:
  - Dynamic scheduling + multiple issue + speculation
- Two approaches:
  - Assign reservation stations and update pipeline control table in half clock cycles
    - Only supports 2 instructions/clock
  - Design logic to handle any possible dependencies between the instructions
  - Hybrid approaches
- Issue logic can become bottleneck

Overview of Design



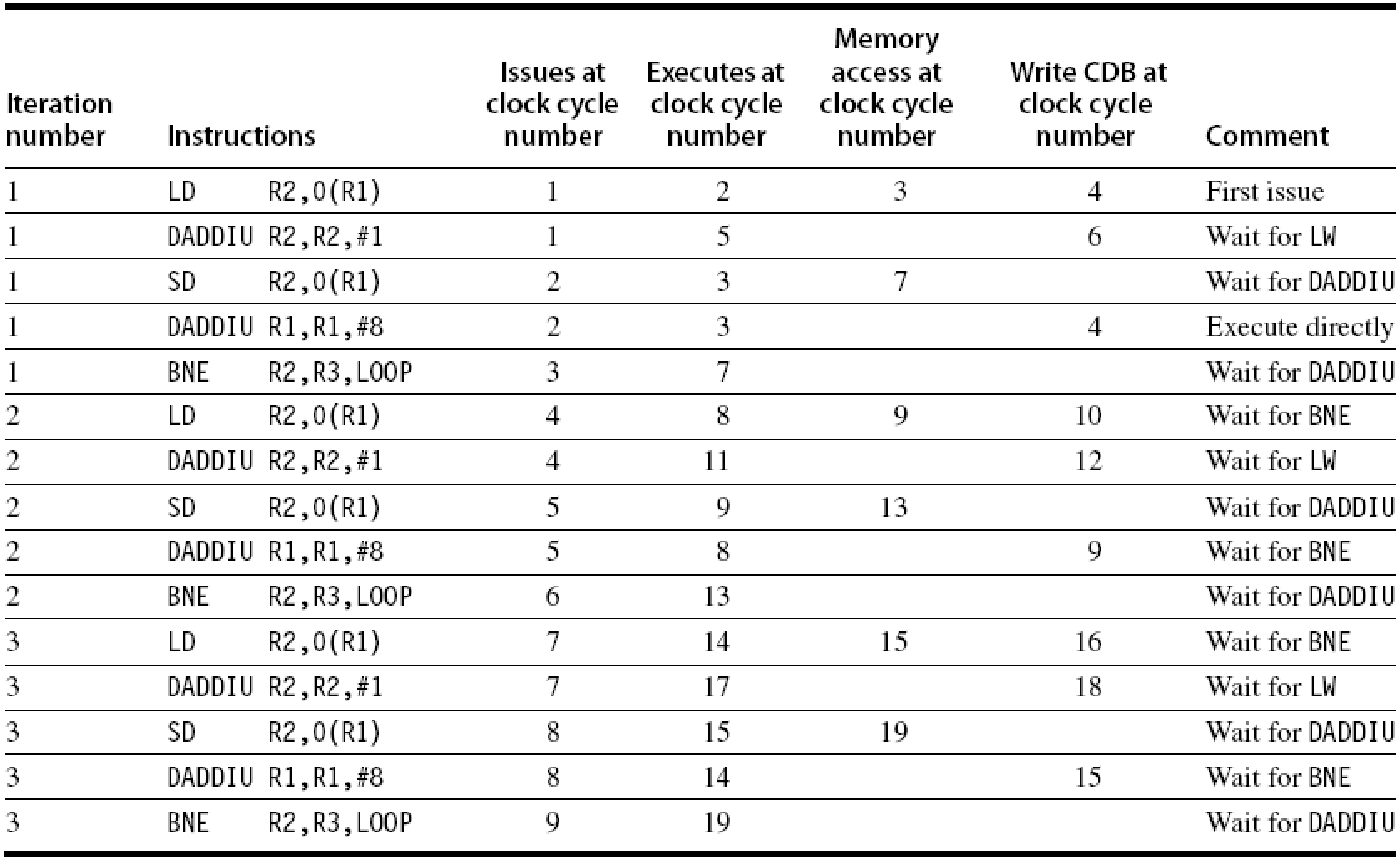
Multiple Issue

- Limit the number of instructions of a given class that can be issued in a “bundle”
  - I.e. one FP, one integer, one load, one store
- Examine all the dependencies among the instructions in the bundle
- If dependencies exist in bundle, encode them in reservation stations
- Also need multiple completion/commit

Example

```
Loop:  LD R2,0(R1)           ;R2=array element
       DADDIU R2,R2,#1       ;increment R2
       SD R2,0(R1)           ;store result
       DADDIU R1,R1,#8       ;increment pointer
       BNE R2,R3,LOOP        ;branch if not last element
```

Example (No Speculation)



Iteration number	Instructions		Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD	R2,0(R1)	1	2	3	4	First issue
1	DADDIU	R2,R2,#1	1	5		6	Wait for LW
1	SD	R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU	R1,R1,#8	2	3		4	Execute directly
1	BNE	R2,R3,LOOP	3	7			Wait for DADDIU
2	LD	R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU	R2,R2,#1	4	11		12	Wait for LW
2	SD	R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU	R1,R1,#8	5	8		9	Wait for BNE
2	BNE	R2,R3,LOOP	6	13			Wait for DADDIU
3	LD	R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU	R2,R2,#1	7	17		18	Wait for LW
3	SD	R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU	R1,R1,#8	8	14		15	Wait for BNE
3	BNE	R2,R3,LOOP	9	19			Wait for DADDIU

Limits to Multi-Issue Machines

- Inherent limitations of ILP
  - 1 branch in 5: How to keep a 5-way VLIW busy?
  - Latencies of units: many operations must be scheduled
  - Need about Pipeline Depth x No. Functional Units of independent operations to keep machines busy,

    e.g. 5 x 4 = 15–20 independent instructions?
- Difficulties in building HW
  - Easy: More instruction bandwidth
  - Easy: Duplicate FUs to get parallel execution
  - Hard: Increase ports to Register File (bandwidth)
    - VLIW example needs 7 read and 3 write for Int. Reg.

      & 5 read and 3 write for FP reg
  - Harder: Increase ports to memory (bandwidth)
  - Decoding Superscalar and impact on clock rate, pipeline depth?

Hprcse

Limits to Multi-Issue Machines

- Limitations specific to either Superscalar or VLIW implementation
  - Decode issue in Superscalar: how wide practical?
  - VLIW code size: unroll loops + wasted fields in VLIW
    - IA-64 compresses dependent instructions, but still larger
  - VLIW lock step => 1 hazard & all instructions stall
    - IA-64 not lock step? Dynamic pipeline?
  - VLIW & binary compatibility is practical weakness as vary number FU and latencies over time
    - IA-64 promises binary compatibility

Studies of the Limitations of ILP - The Hardware Model

- Ideal processor
  - all artificial constraints on ILP are removed.
- Register renaming
  - There are an infinite number of virtual registers available
    - Architecturally visible registers
  - all WAW and WAR hazards are avoided
  - an unbounded number of instructions can begin execution simultaneously
- Branch prediction
  - Branch prediction is perfect
  - All conditional branches are predicted exactly
- Jump prediction
  - All jumps are perfectly predicted
  - including jump register used for return and computed jumps
  - an unbounded buffer of instructions available for execution.
- Memory-address alias analysis
  - All memory addresses are known exactly
  - a load can be moved before a store if the addresses are not identical.

Studies of the Limitations of ILP - The Hardware Model

- The Hardware Model
  - can issue an unlimited number of instructions at once
  - all functional unit latencies are assumed to be one cycle perfect caches
    - all loads and stores always complete in one cycle (100% hit).
  - ILP is limited only by the data dependences
- ILP available in a perfect processor
  - Average amount of parallelism available

Studies of the Limitations of ILP

- The perfect processor must do
  - Look arbitrarily far ahead
    - to find a set of instructions to issue
    - predicting all branches perfectly.
  - Rename all register uses
    - to avoid WAR and WAW hazards.
  - Determine data dependencies among the instructions
    - if so, rename accordingly.
  - Determine memory dependences
    - handle them appropriately.
  - Provide enough replicated functional units
    - to allow all the ready instructions to issue

Studies of the Limitations of ILP

Limitations on the Window Size and Maximum Issue Count

- The instruction window
  - The set of instructions that are examined for simultaneous execution
    - limits the number of instructions that begin execution in a given cycle
  - limited by the required storage, the comparisons, and a limited issue rate
    - In the range of 32 to 126
- Real processors more limited by
  - number of functional units
  - numbers of buses
  - register access ports
- large window sizes are impractical and inefficient

Superscalar v. VLIW

Superscalar	VLIW
Smaller code size	Simplified Hardware for decoding, issuing Instructions
Binary compatability across generations of hardware	No Interlock Hardware (compiler checks?)
	More registers, but simplified Hardware for Register Ports (multiple independent register files?)