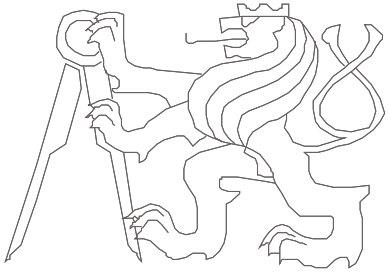
Advanced Computer Architectures

04

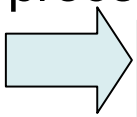
Superscalar Techniques – Instruction prefetching (Branch prediction etc.)



Czech Technical University in Prague, Faculty of Electrical Engineering
Slides authors: Michal Štepanovský, update Pavel Píša

Superscalar Techniques…

- The goal is to achieve a **maximum throughput of the instruction processing**
- Instruction processing can be analyzed as instructions flow or data flow, more precisely:
  - register data flow – data flow between processor registers
  - **instruction flow** through the pipeline
  - memory data flow – to/from memory

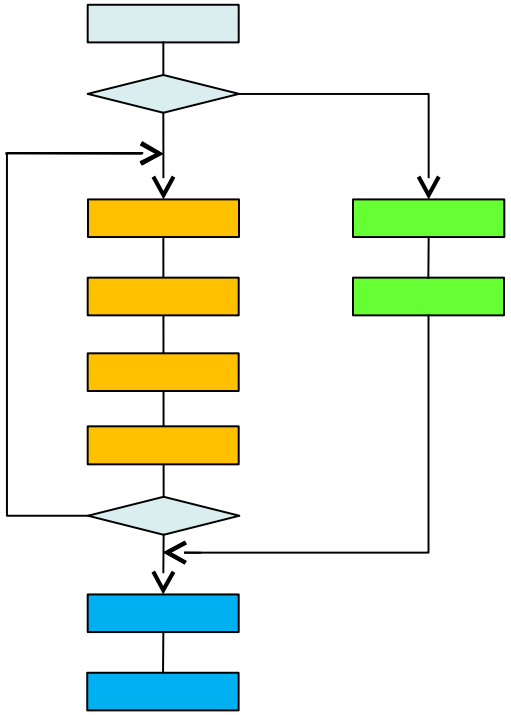


Today's lecture topic

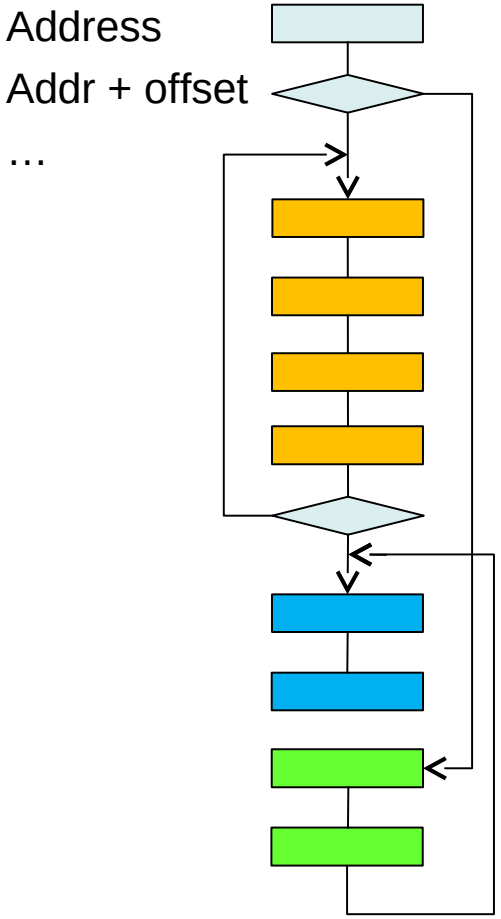
- It roughly matches to:
  - Arithmetic-logic (ALU) and other computational instructions (FP, bit-field, vector) processing
  - Branch instruction processing
  - Load/store instruction processing
- maximizing the throughput of these three flows (or complete flow) correspond to the minimizing penalties and latencies of above three instructions types

Control Flow Graph

The program described as a Control Flow Graph (CFG):



CFG has to be mapped into sequential memory:

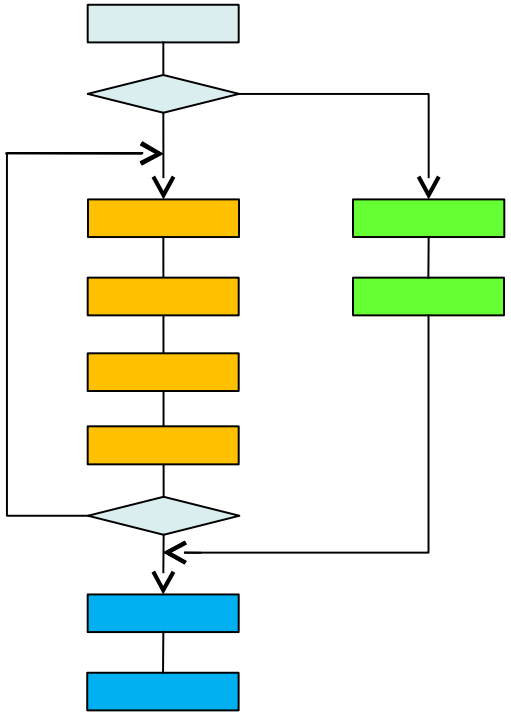


And what about object-oriented programming???

Virtual methods ???

Control Flow Graph

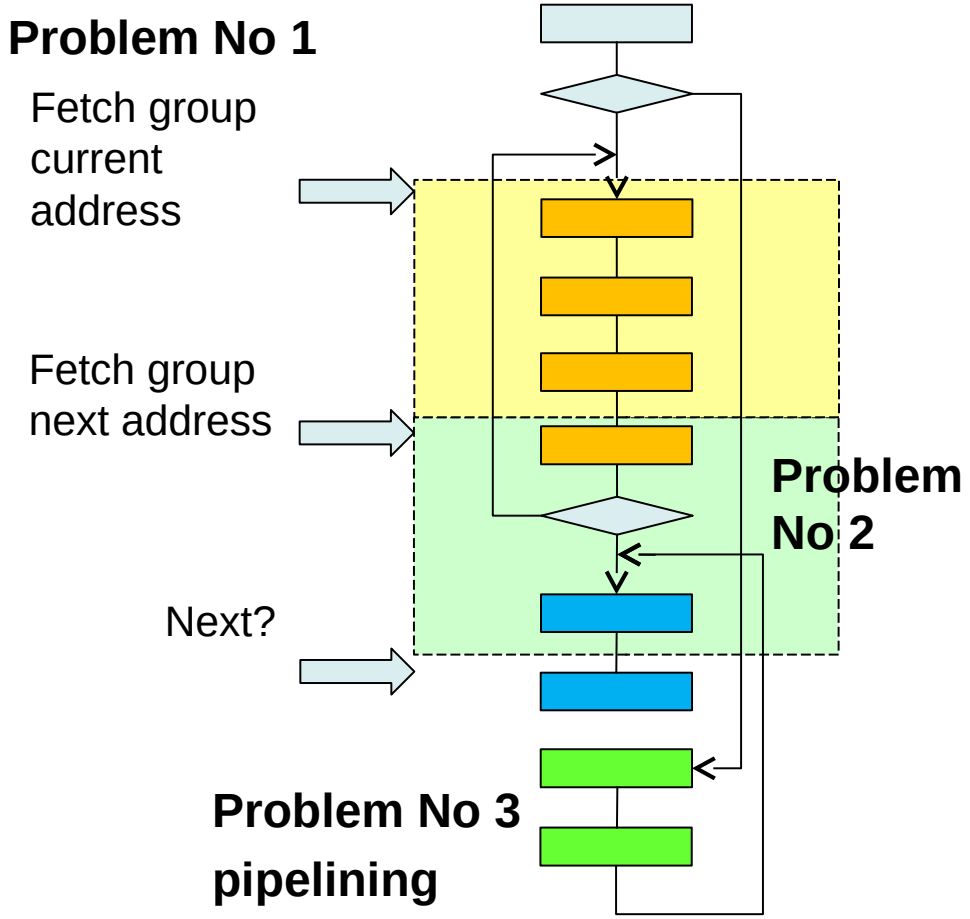
The program described as a Control Flow Graph (CFG):



And what about object-oriented programming???

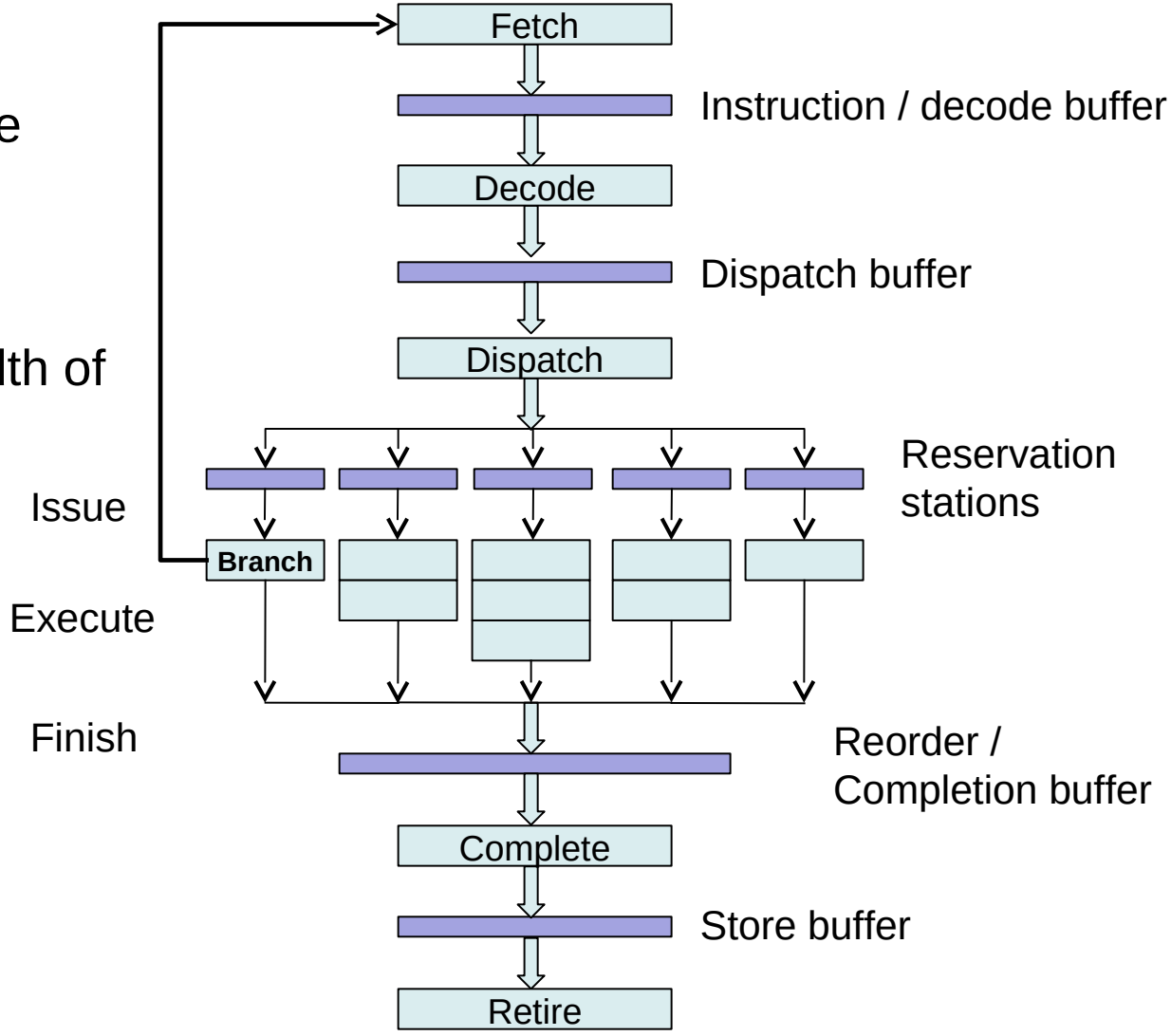
Virtual methods ???

CFG has to be mapped into sequential memory:

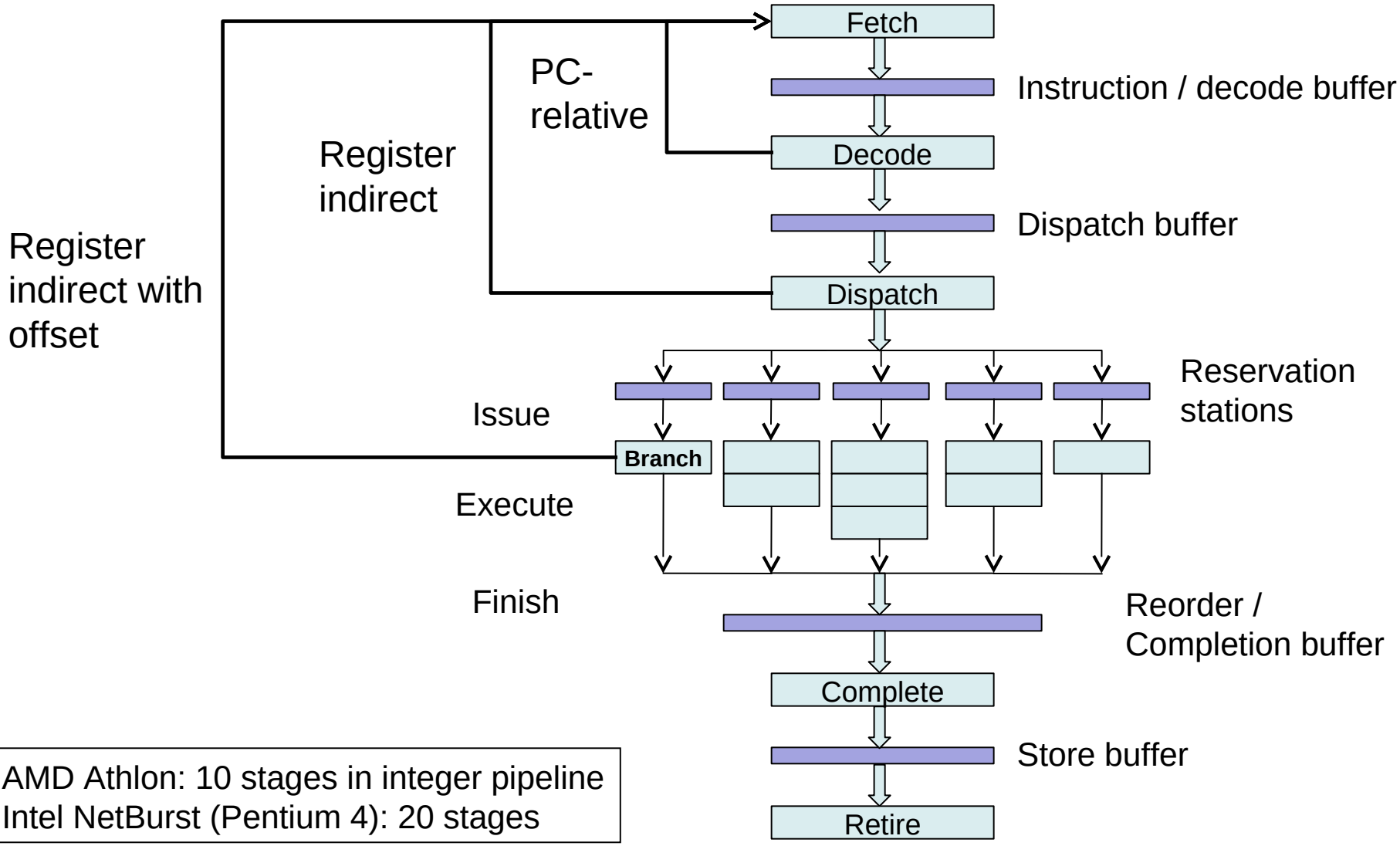


Branch prediction – Motivation

- the penalty of three cycles in fetching the next instruction;
- number of empty instruction slots multiplied by the width of the superscalar machine;
- Amdahl's law..



Branch prediction – Motivation



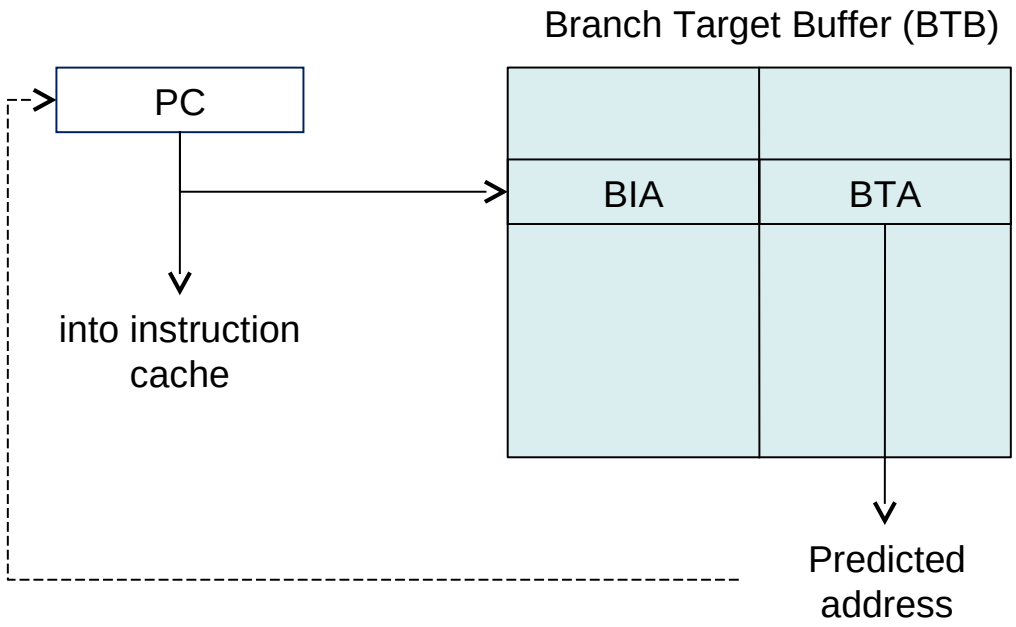
AMD Athlon: 10 stages in integer pipeline
Intel NetBurst (Pentium 4): 20 stages

Branch prediction

- Two fundamental components:
  - branch target speculation (where is next instruction),
  - branch condition speculation (if the branch is taken).

- Branch target speculation:
  - BTB (Branch Target Buffer) – cache (associative memory) with two fields: BIA (Branch Instruction Address) and BTA (Branch Target Adress) – accessed during the instruction fetch using the instruction fetch address (PC)
  - When BIA matches with current PC, the corresponding BTA is accessed and if the branch instruction is predicted to be taken, BTA is used to modify PC

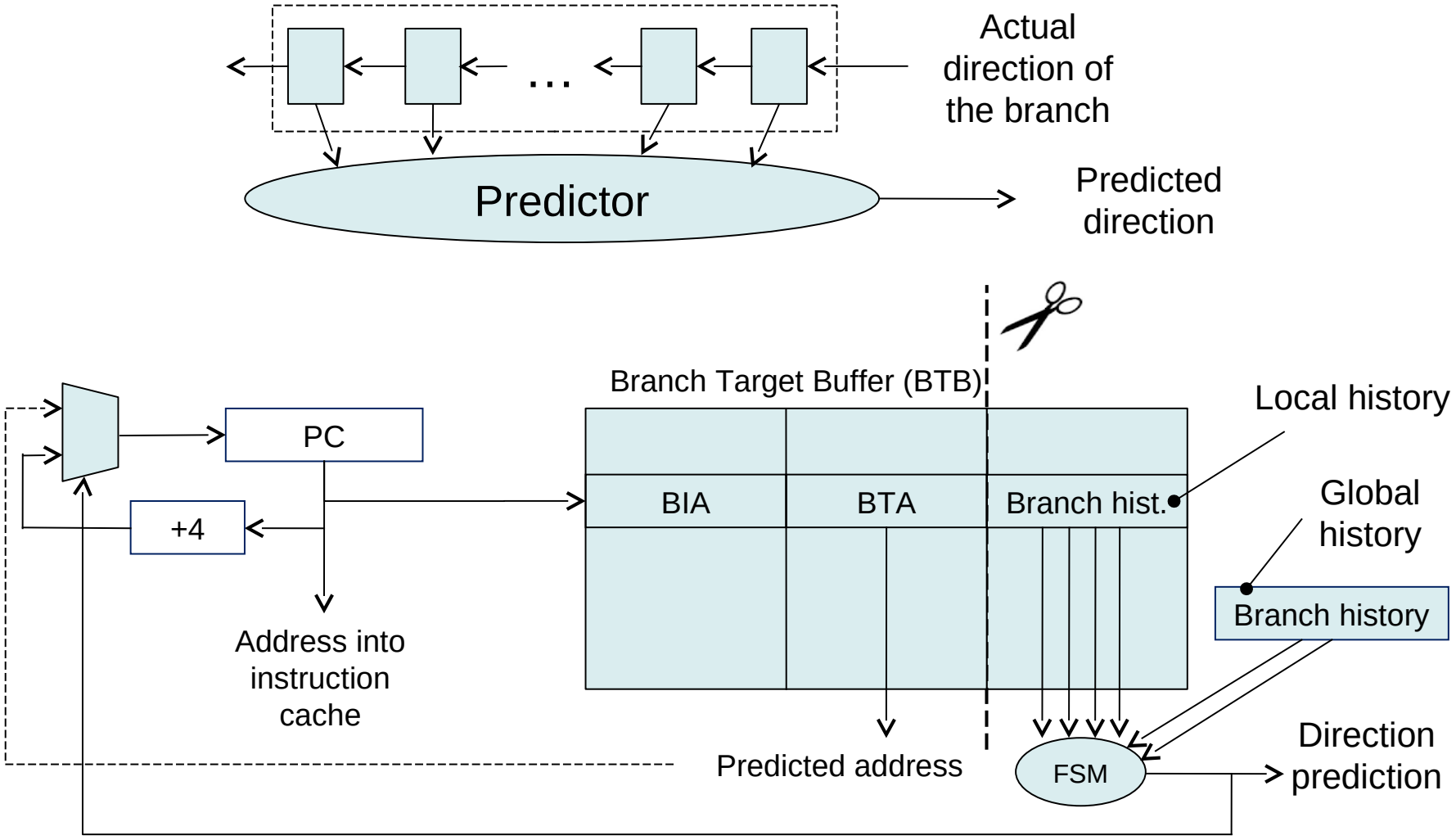
Branch Target Speculation



Branch prediction

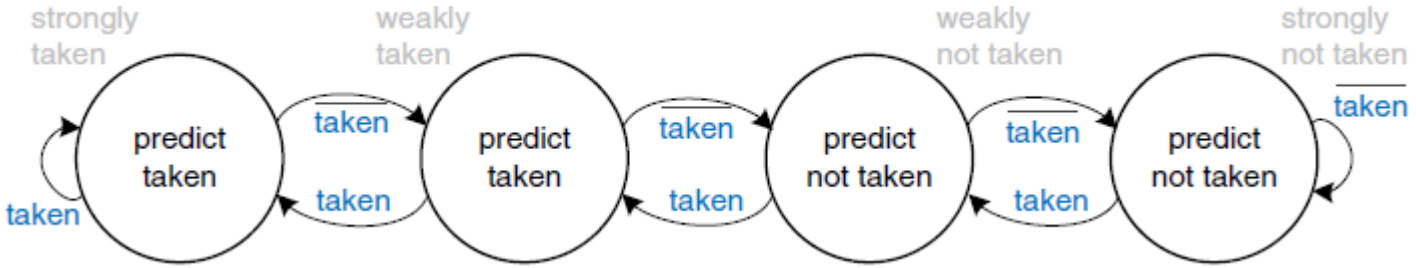
- Branch condition speculation (taken/not-taken):
  - Static prediction (70%-80%)
    - BTFNT (Backwards Taken / Forwards Not-Taken) – for, while, do-while,.. - relatively to PC, branch delay slot… the compiler places unlikely branches after likely path through function.
    - Heuristics used to analyze the program (NULL pointer, equality, inline functions…) – branch hints coded in branch instructions (if supported by ISA)
    - Profilation – runs with different inputs - statistics
  - Dynamic prediction (80%-97%)
  - Hybrid (combination of static and dynamic prediction. Static prediction in the initial phase, when the dynamic prediction is not available, the first pass through code or when prediction slot is reused for later PC address).

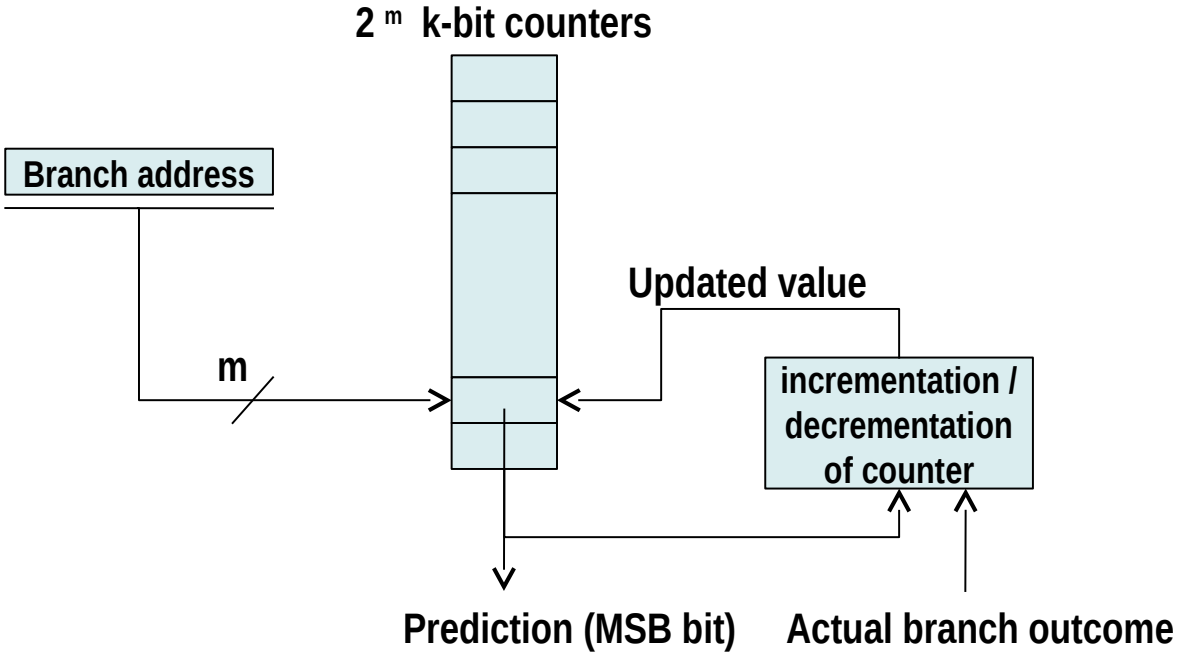
Branch prediction



Branch condition prediction – local predictor

- Smith's algorithm (Saturating counter)





Branch condition prediction – local predictor

- Smith's algorithm

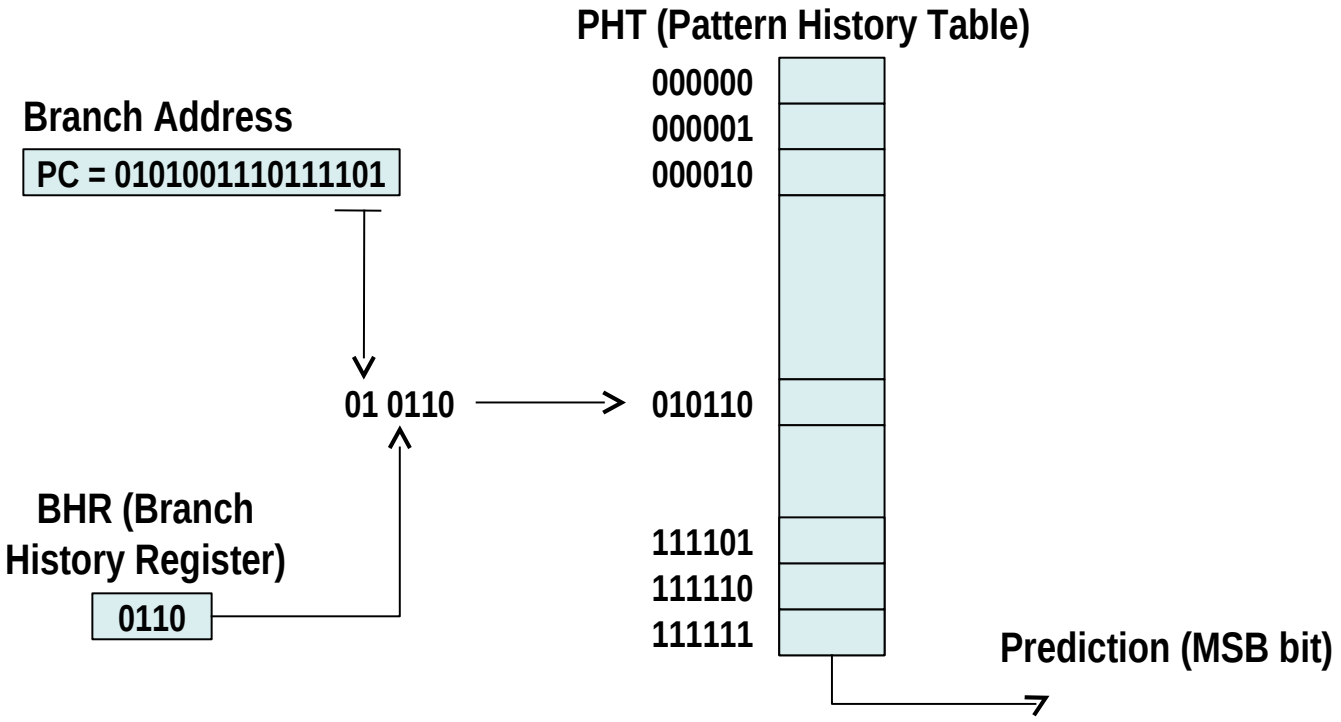
Problem ???

If Program Counter is 32 bits wide, we will need (1/8)*k*2^32 B of memory to store the state of all saturating counters (if k==2, we need 1GB of memory), much smaller n-way associative branch history table is used.

- **Conflict aliasing**:
  - Neutral interference
  - Negative interference
- Compulsory aliasing in addition – we will see it later… (because of indexing by address-history combination)

Branch condition prediction – global predictor

- **Global-History** Two-Level Branch Predictor with a 4-bit Branch History Register



What is the optimal number of bits for BHR a for BA?

Branch condition prediction – global predictor

- **Global-History** Two-Level Branch Predictor with a 4-bit Branch History Register

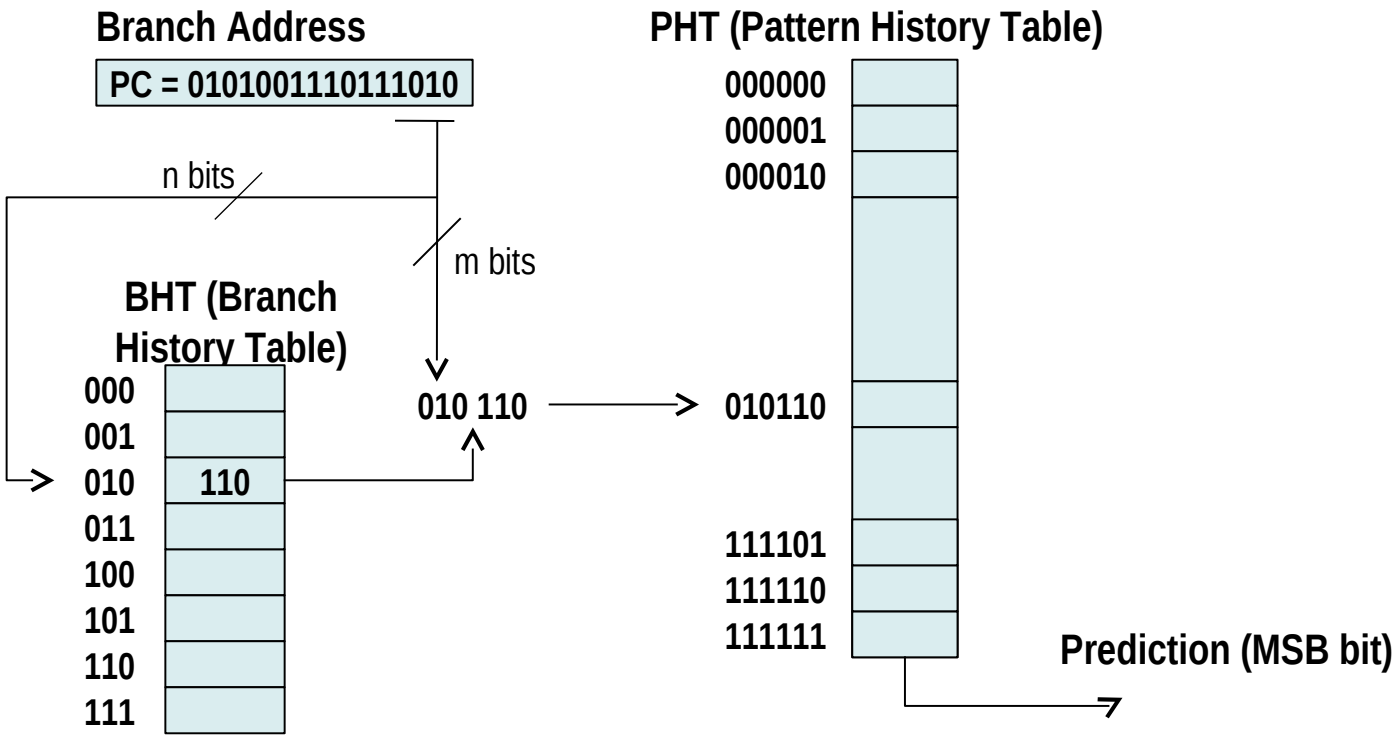
- Why use global history for PHT indexation?

```
a=0;
if(condition #1)   a=3;
if(condition #2)   b=10;
if(a <= 0)  F();
```

- The behavior of a branch may be connected (or correlated) with a different branches conditions evaluation in the past.
- In our example, execution of function F() depends on the condition #1. The condition #2 is irrelevant. Predictor must be able to learn this behavior (distinguish these branch conditions).

Branch condition prediction

- **Local-History** Two-Level Predictor with a 3-bit Branch History Table



Intel P6 uses 4 bits for BHR

Branch condition prediction

- **Local-History** Two-Level Predictor with a 3-bit Branch History Table

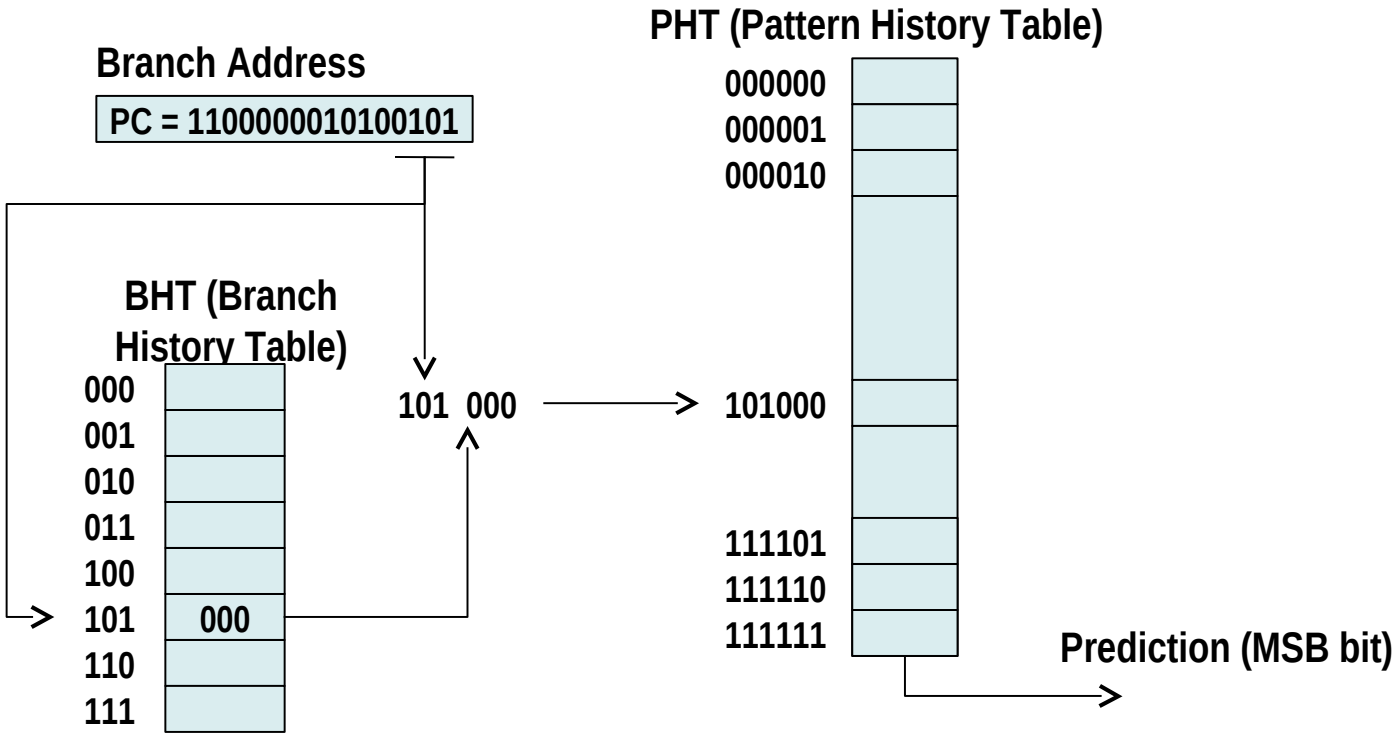
- Why local history for PHT indexation?

```
do{
...
}while(condition);
```

- Because the behavior of a branch may be closely related to its own history…

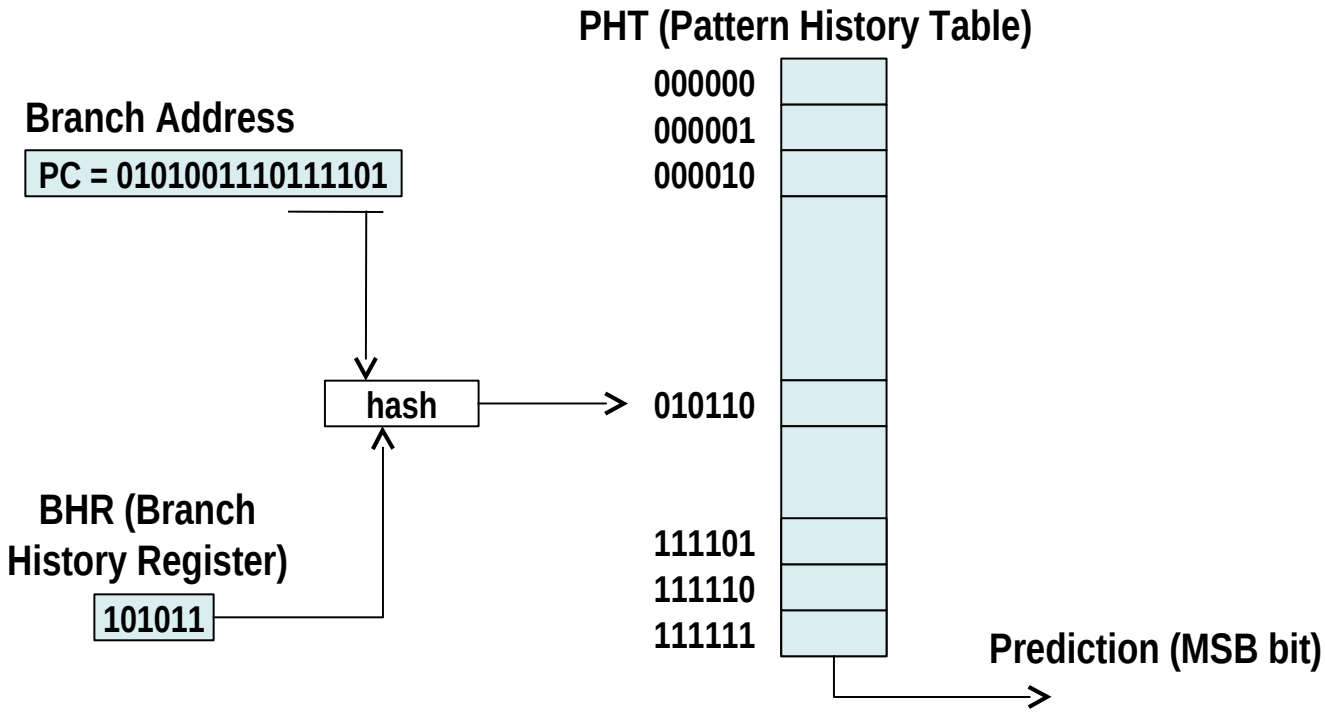
Branch condition prediction

- **Local-History** Two-Level Predictor with a 3-bit Branch History Table – <u>An example:</u>
- Let's suppose that at the address 0xC0A5 is the „loop-closing branch" with the pattern: 1110111010111011101..., where 1 means taken branch. How many cycles needs the predictor to learn this pattern? Initially, BHT and PHT contain zeros.



Branch condition prediction

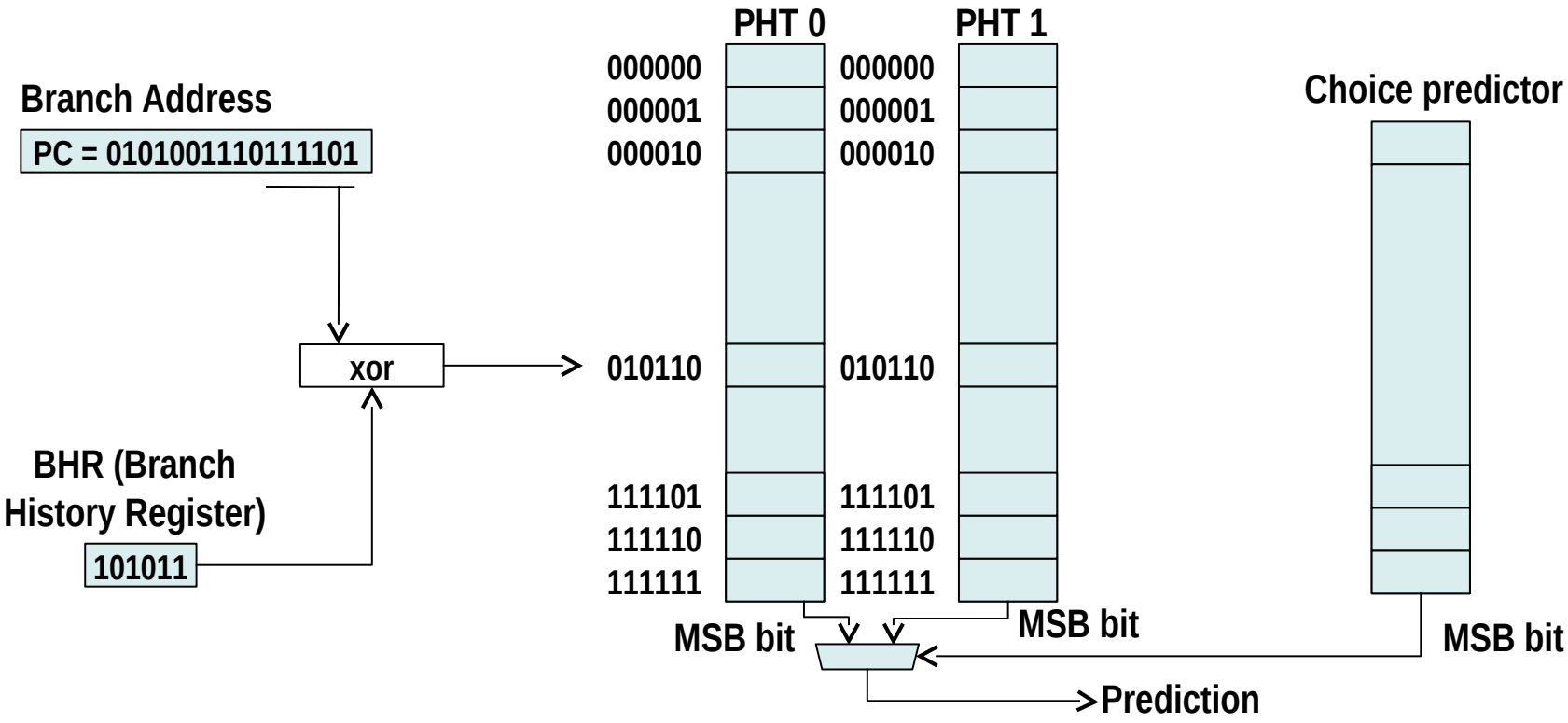
- **Index-sharing predictors**… (hash of BHR and PC) – better use of the index bits (it tends to contain more information due to the nonuniform distribution of PC values and branch histories)
- For example, **gshare predictor**:



gshare: Hash: XOR

Branch condition prediction

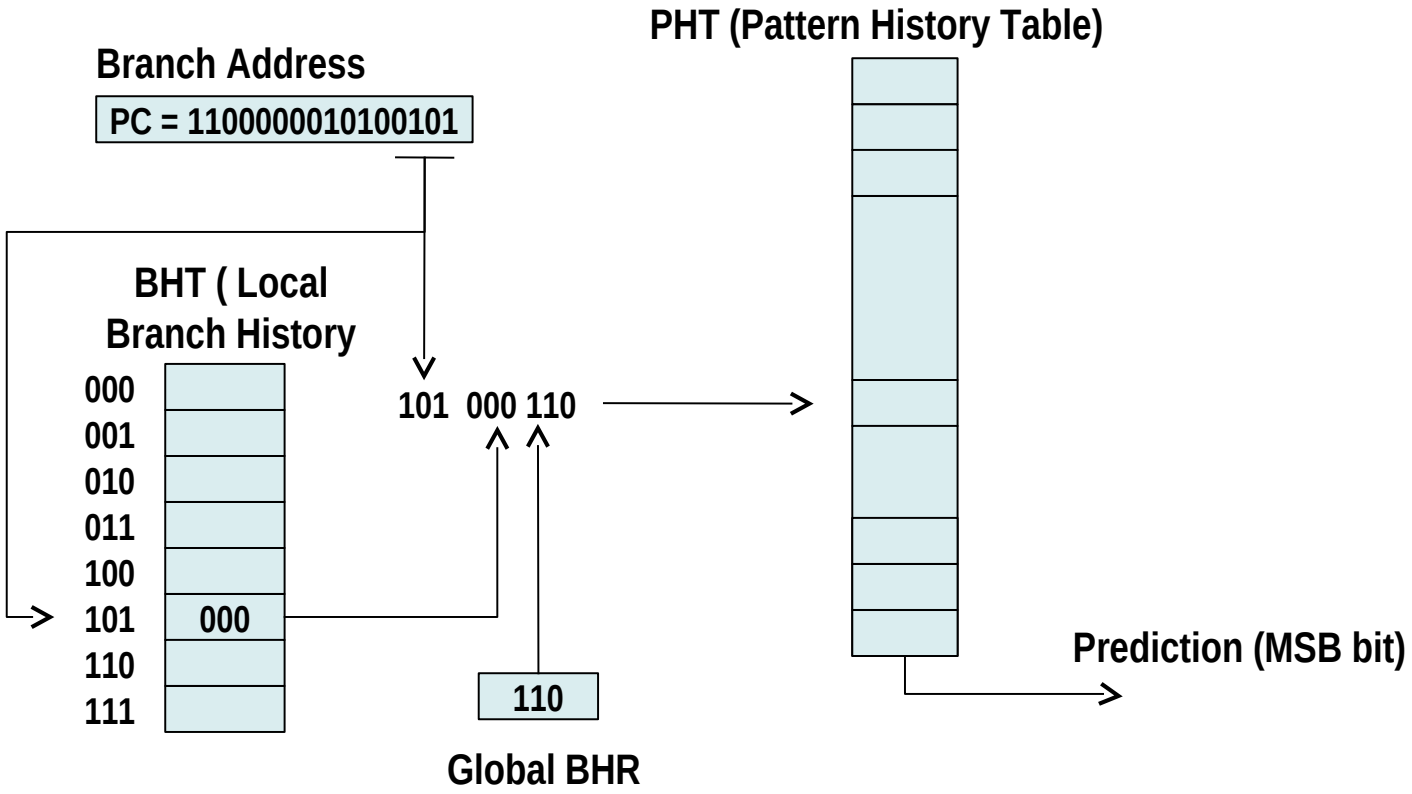
- **Index-sharing predictors**…
- bi-Mode predictor – two separated PHT – the same index or hash:



- Choice predictor – Smith. The branches with a taken bias are placed in PHT1 and other ones into PHT0. Negative interference => neutral interference. (the statistic of branches has this property)

Branch condition prediction

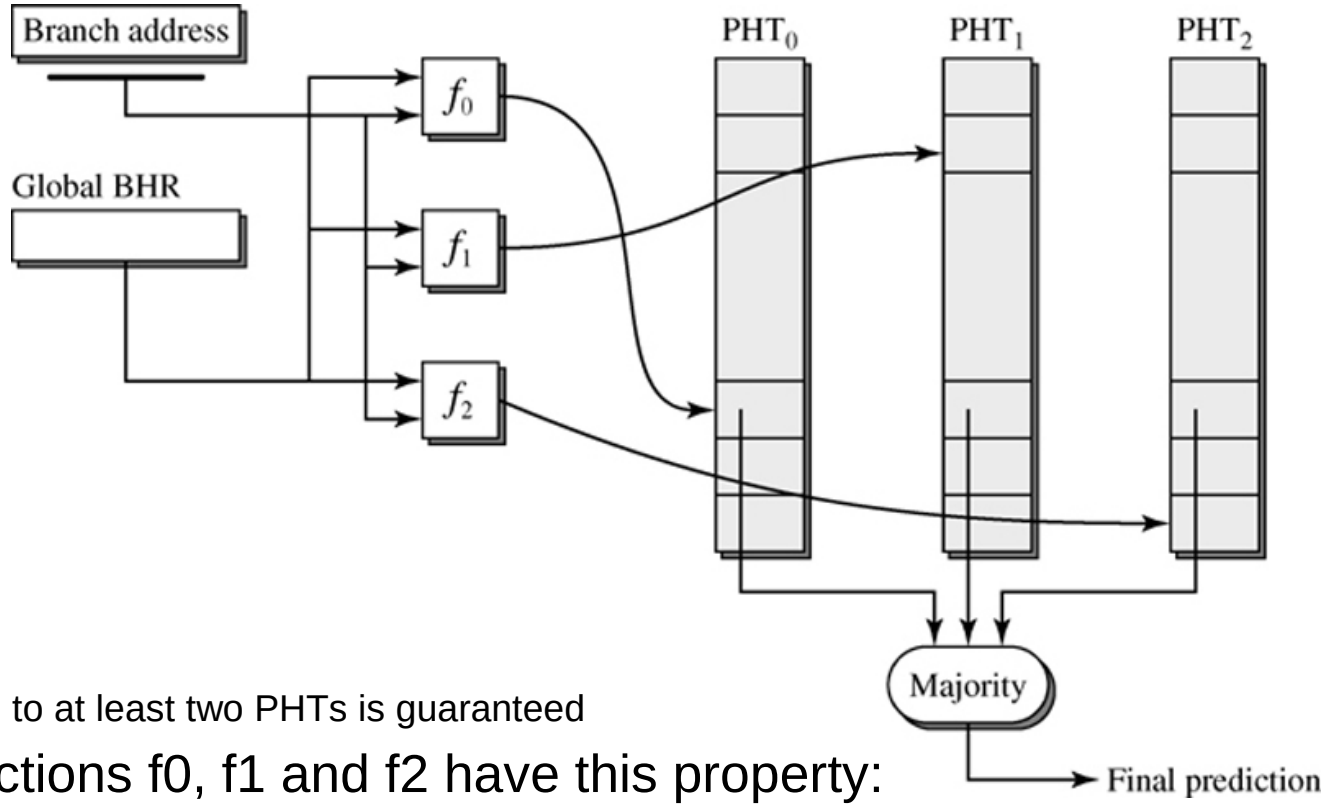
- **alloyed predictor**



- Combination of PC, local history and global history as an index into PHT

Branch condition prediction

- **gskewed predictor**

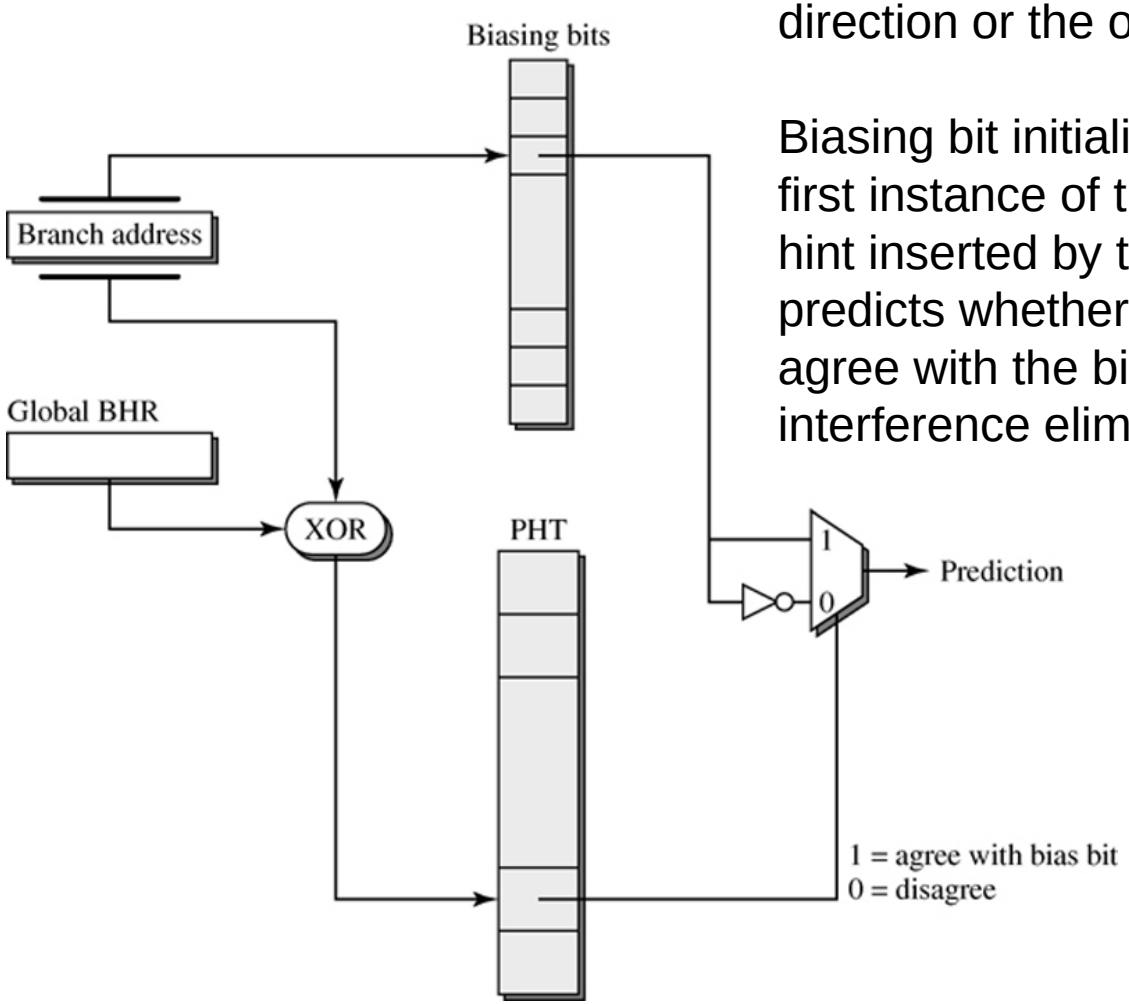


Non-conflict access to at least two PHTs is guaranteed

- Hash functions f0, f1 and f2 have this property:
  if f0(x1)=f0(x2) for x1≠x2, then f1(x1) ≠ f1(x2) and f2(x1) ≠ f2(x2)
- **Total update** (update of all banks by branch outcome) and
  partial update (better) – does not update a bank if that particular bank mispredicted, but the overall prediction was correct.

Branch condition prediction

- **agree predictor**

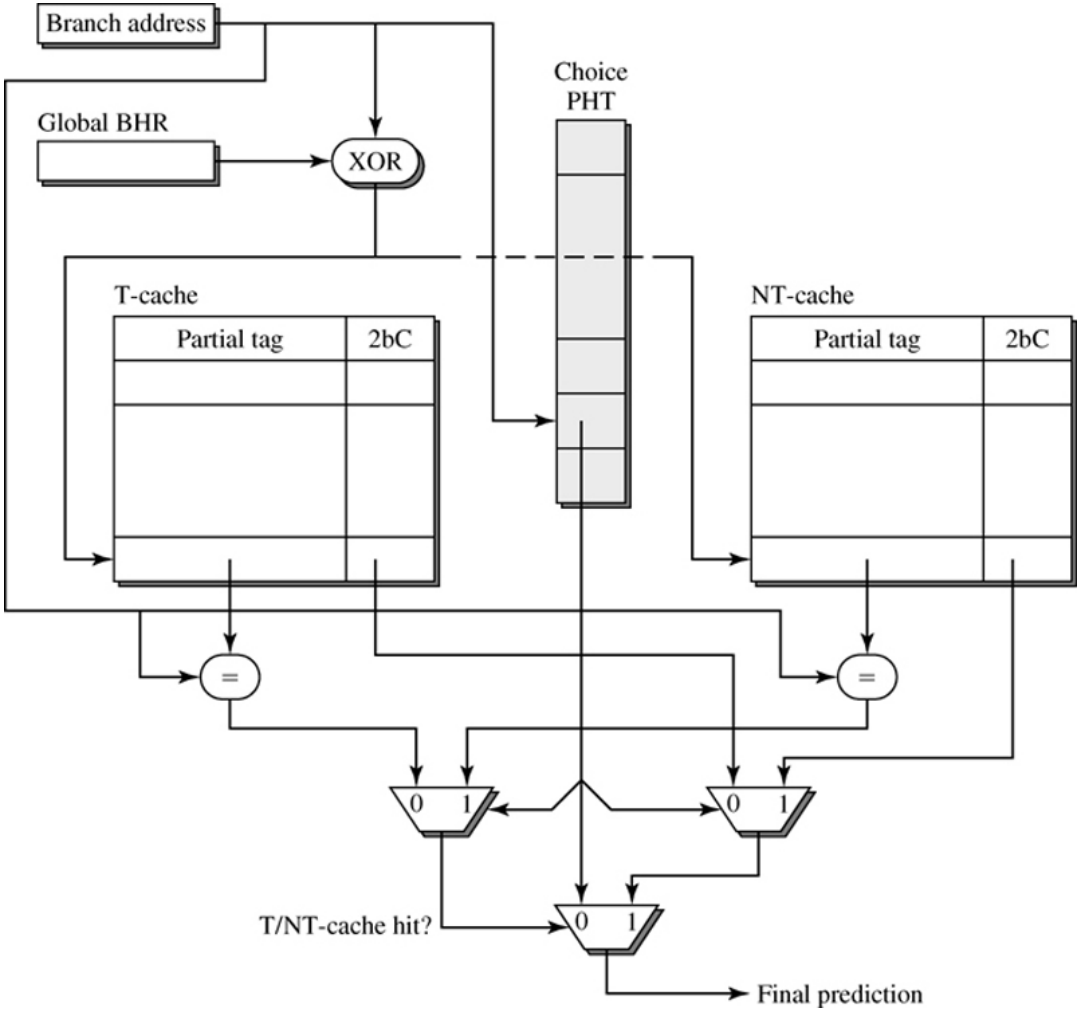


Branches tend to be heavily biased in one direction or the other (taken/not-taken).

Biasing bit initialized to the outcome of the first instance of the branch, or a branch hint inserted by the compiler. PHT predicts whether the branch outcome will agree with the biasing bit. (Negative interference elimination)

Branch condition prediction

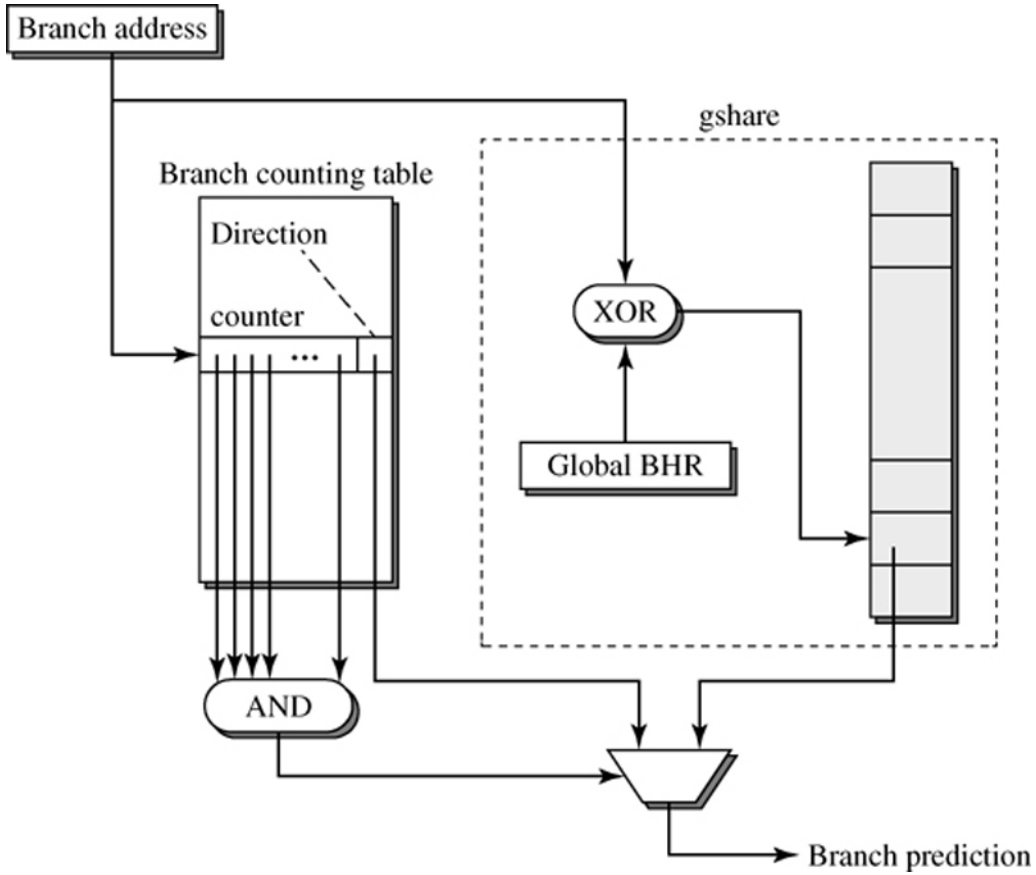
- **YAGS predictor**



Extension of Bi-mode predictor… but with the significant difference!
If PHT indicates “taken”, then “Not-Taken” (NT) cache is used for prediction. Both caches store only the branches which do not agree with PHT…

Branch condition prediction

- **Branch Filtering predictor**



Branches tend to be heavily biased in one direction or the other. Expensive HW structures are not required for these saturating branches and can be used only for more complicated ones.

If the counter in BCT has been incremented to its max value, then this branch will no longer update the PHT.

If BCT makes miss-prediction, then the counter is reset.

Branch condition prediction

Pattern-Based Predictors

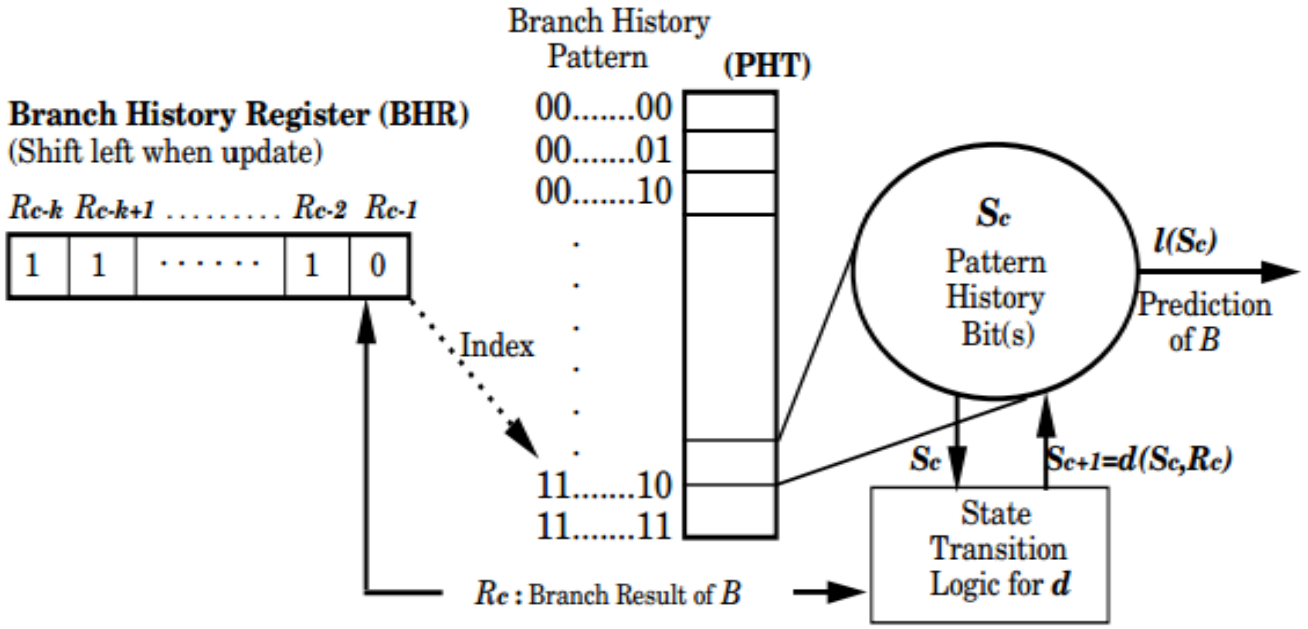
- They extend the idea of 1-bit or in general n-bit predictors to recognize specific branch pattern
- Example. Consider these patterns (0-not taken branch, 1-taken):
  0101010101, 110110110, 00111010011101, …
  These patterns are hard to predict (by 1 or 2-bit predictors) - the bias of the branch develops in time…

- **0101010101** – after 0 comes 1, after 1 comes 0
- **110110110** – after 11 comes 0, after 10 comes1, after 01 comes 1
- **00111010011101** – after 100 comes 1, after 001 comes 1, after 011 comes 1, after 111 comes 0, after 110 comes 1, after 101 comes 0, after 010 comes 0

- Did previous predictors the prediction on these patterns well ???
- Note: All predictors which use the combination of local and global history are called correlating predictors.

Branch condition prediction

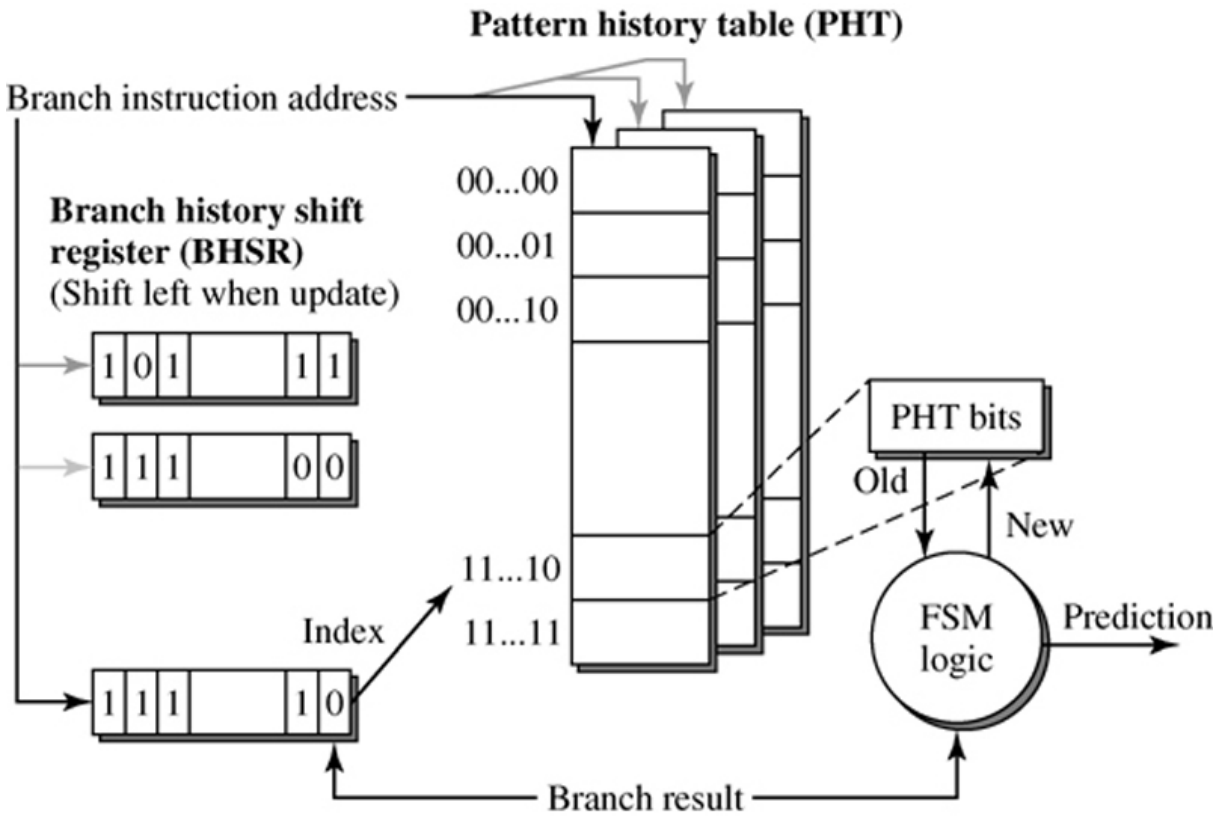
- **Two-level adaptive branch predictor** (**Correlating/ed predictor**)
- Authors: Yeh and Patt from University of Michigan



- First level – history of last K branches (for example last K branch instructions or last K resolutions of single instruction) – builds the pattern
- Second level – the behavior of last N appearances of the pattern indexes PHT
- BHR is associated with given branch instruction

Branch condition prediction

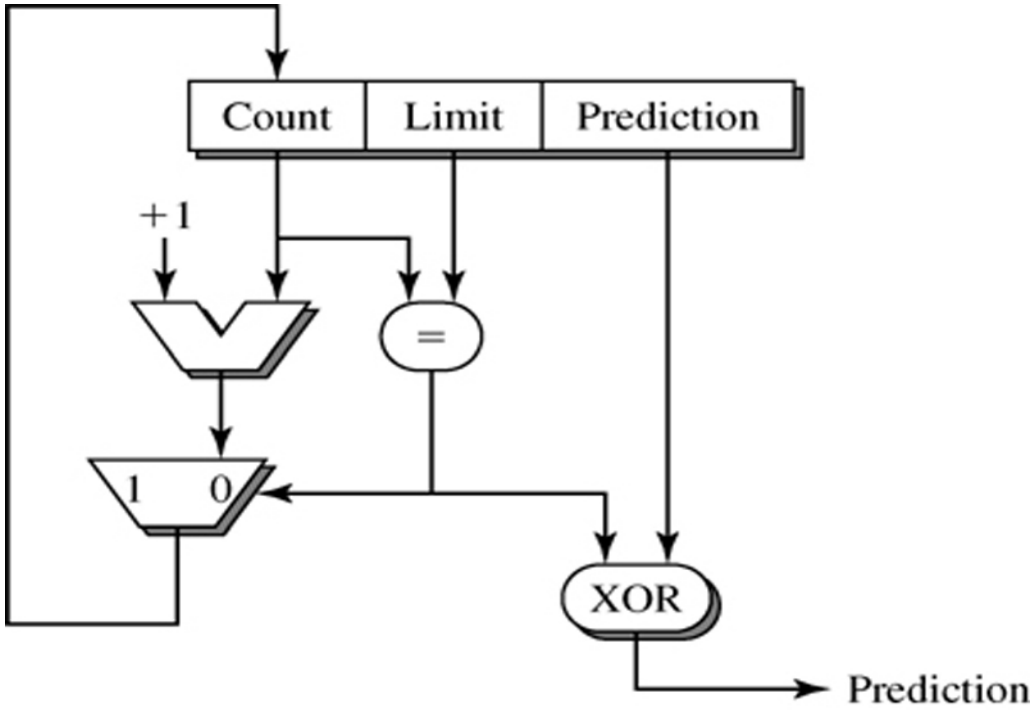
- **Two-level adaptive branch predictor** (**Correlating/ed predictor**)
- Authors: Yeh and Patt from University of Michigan
- The “same” picture:
  (Intel P6)



- Sometimes, PHT is drawn as a matrix, where rows (columns) are indexed by PC and columns (rows) are indexed by BHR. Global BHR may also be used

Branch condition prediction

- **Loop Counting predictor**
- Many iterations
- Only in combination with other predictors (see hybrid predictors)
- Pentium M



Branch condition prediction

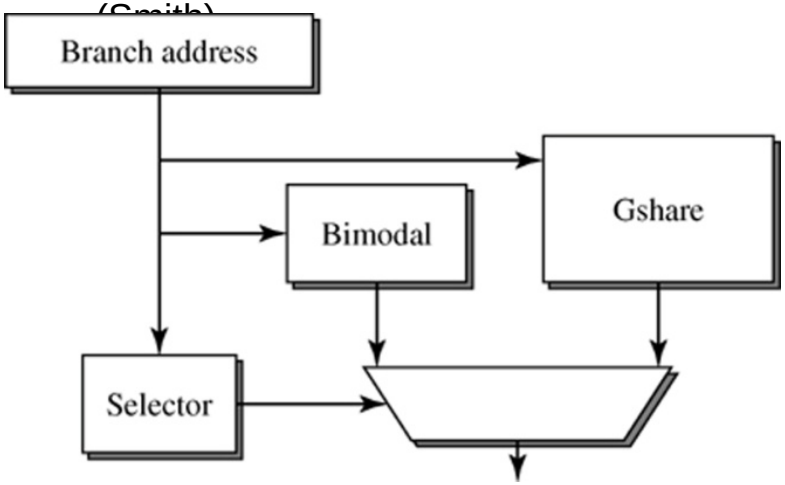
- Advanced predictors take into account negative (and neutral) interference (bi-Mode predictor and others …)

Some other dynamic predictors:

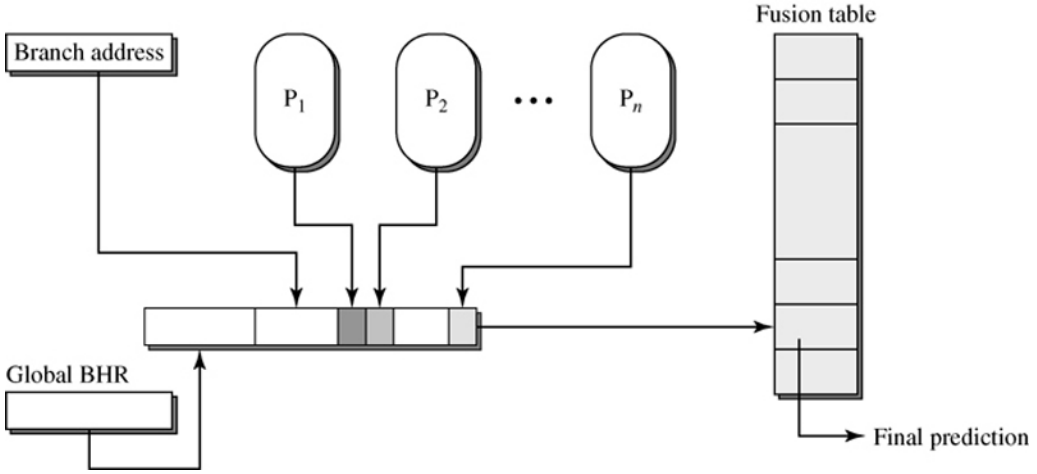
- Perceptron predictor (longer history, correlated branches detected)
- Data flow predictor – explicitly tracks inter-register dependences

Hybrid, multi-hybrid and fusion-based predictors:

- Tournament predictor – two predictors P0 and P1 and meta-predictor M (Smith)



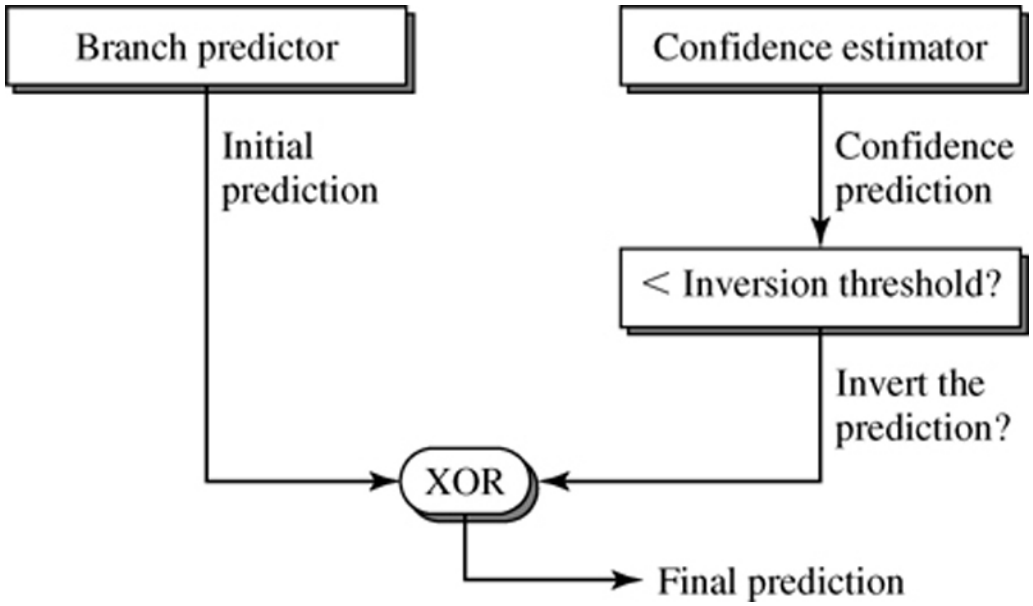
(a) Hybrid: Tournament predictor in this example uses Bimodal and Gshare predictor



(b) Fusion-Based Hybrid Predictor – replaces multiplexer (which selected only single predictor) with **Fusion**

Branch condition prediction

- **Selective branch inversion (SBI)**
- Does not focus primarily on interference reduction but to minimise interference impact/consequences
- It evaluates (estimates) confidence of prediction of predictor
- For example: SBI Bi-Mode predictor provides better results than plain Bi-Mode (reasons: interference avoidance + interference correction)



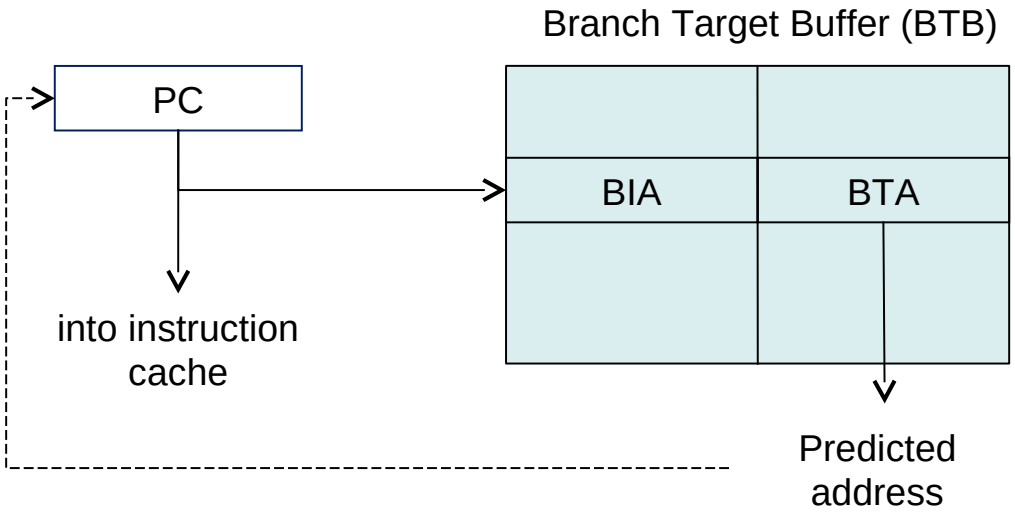
Branch prediction

- Two fundamental components:
  - branch condition speculation (if),
  - branch target speculation (where)

(from slide 10 till now)

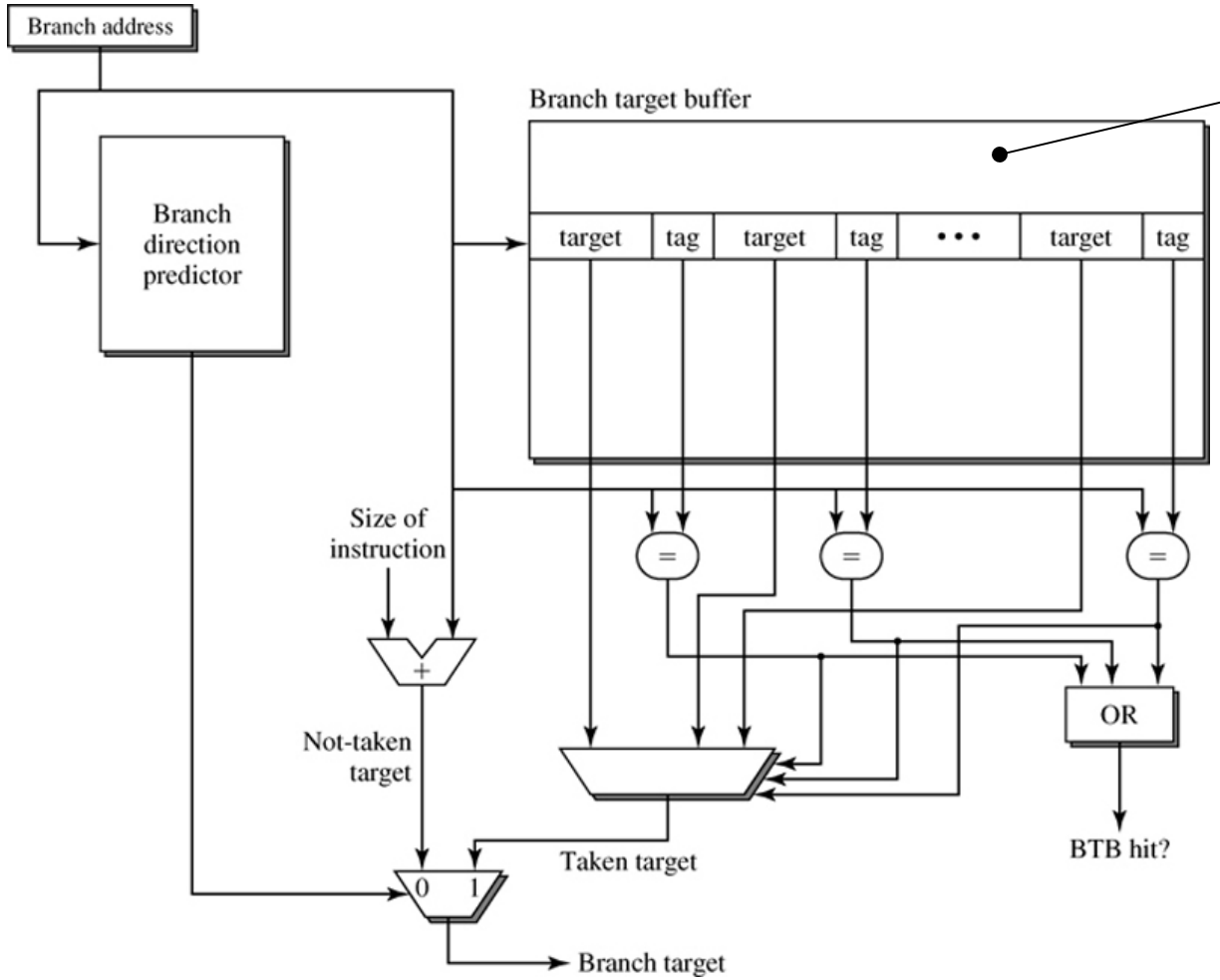
The topic from now

- Branch target speculation:
  - BTB (Branch Target Buffer) – associative cache with two fields: BIA (Branch Instruction Address) and BTA (Branch Target Adress)



Branch **target** speculation

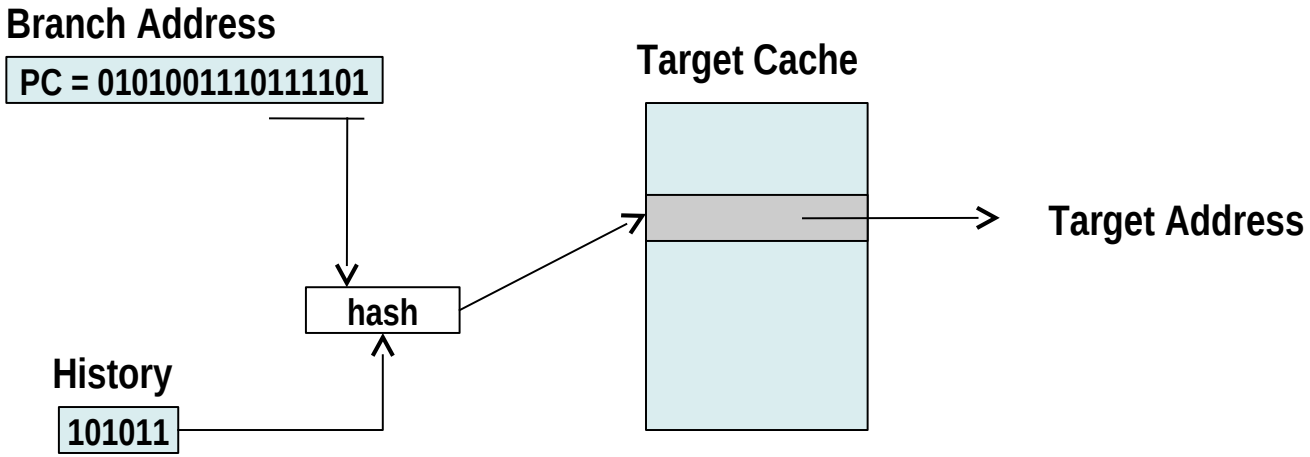
- **Branch Target**: PC-relative branches or indirect branches (run-time



Store the targets of taken branches, cache location or the first target instructions, …
There are more option

Branch **target** speculation – Indirect branches?

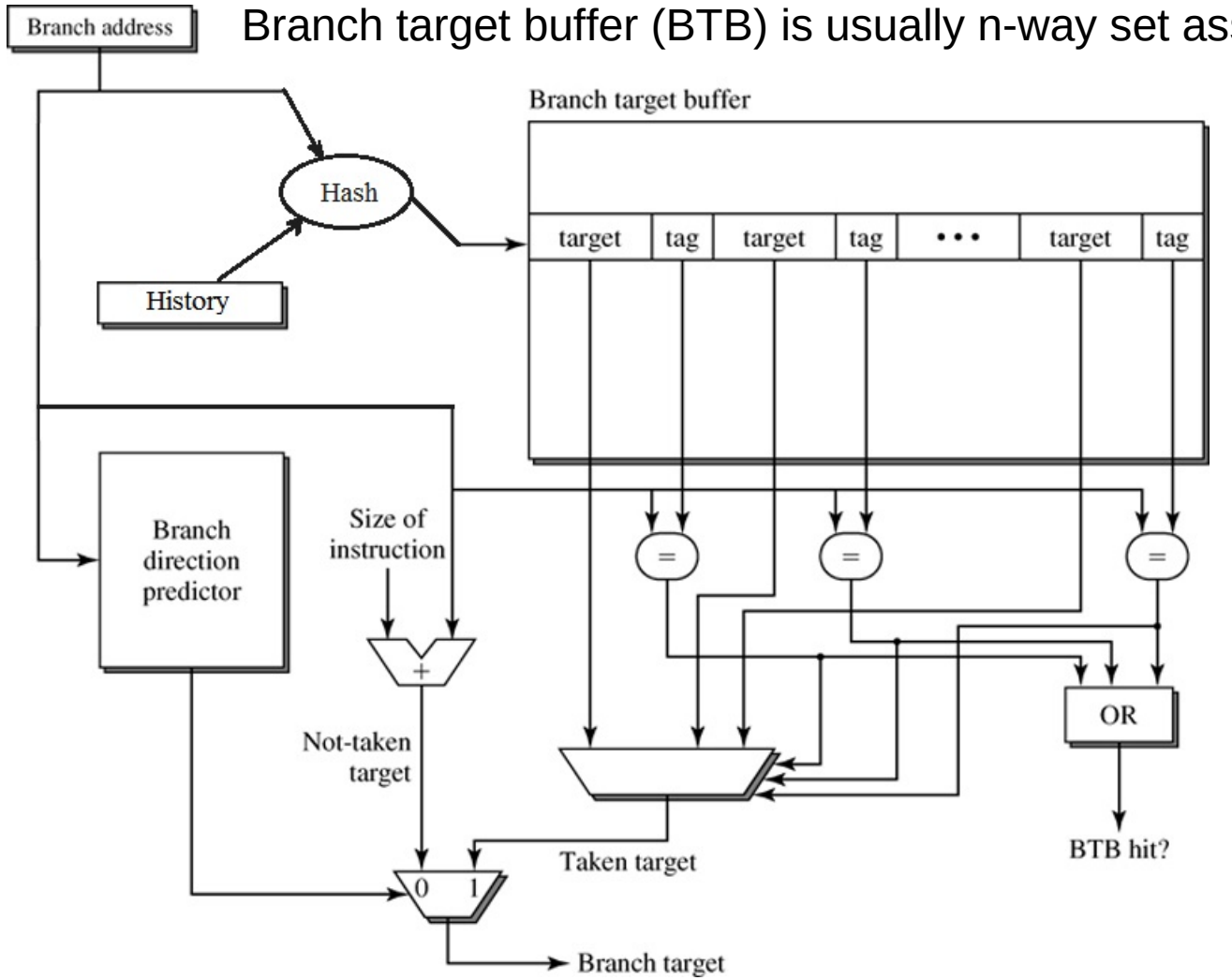
- MIPS: jalr $rX or jr $rX - what it is the target address???
- Object-oriented programming and polymorphism… The result is a lot of indirect branches (in comparison with imperative programming)
  The key idea:



- Indexation of BTB (Branch Target Buffer)
- The hash function can be the “simple” concatenation
- Another point of view: It is the two-level predictor, but the cache stores branch target addresses (not the branch history)

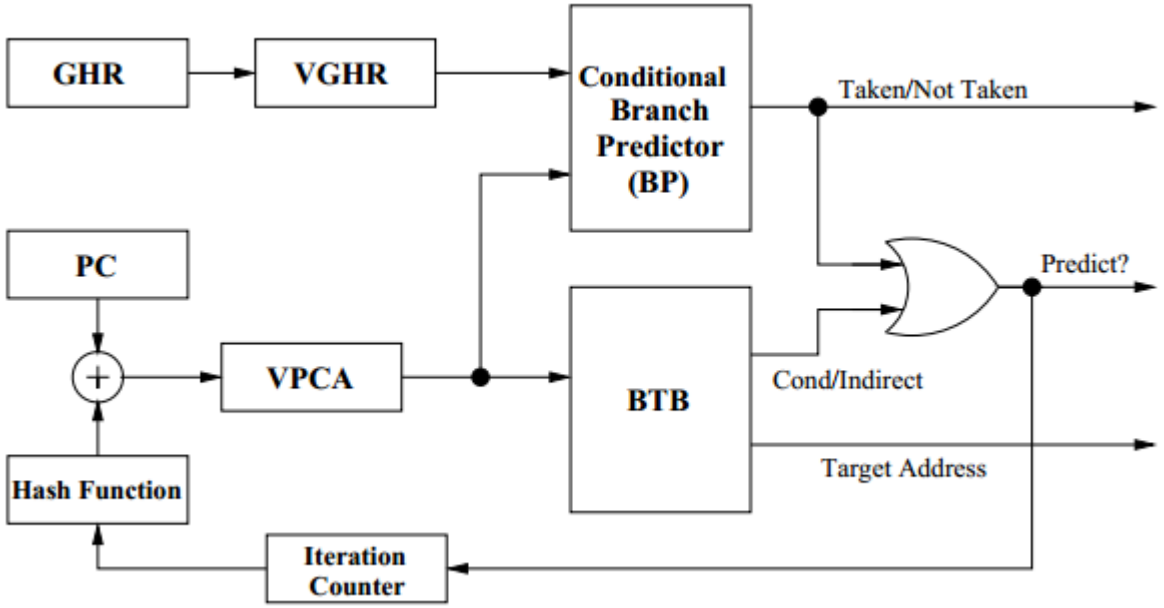
Branch **target** speculation – Collisions in cache…

Branch target buffer (BTB) is usually n-way set associative



Virtual Program Counter (VPC) Prediction

An example:



Instead of expensive HW reuse simple taken/not taken logic which is iterated MAX_ITER times (proposed 4 to 12) for indirect branch

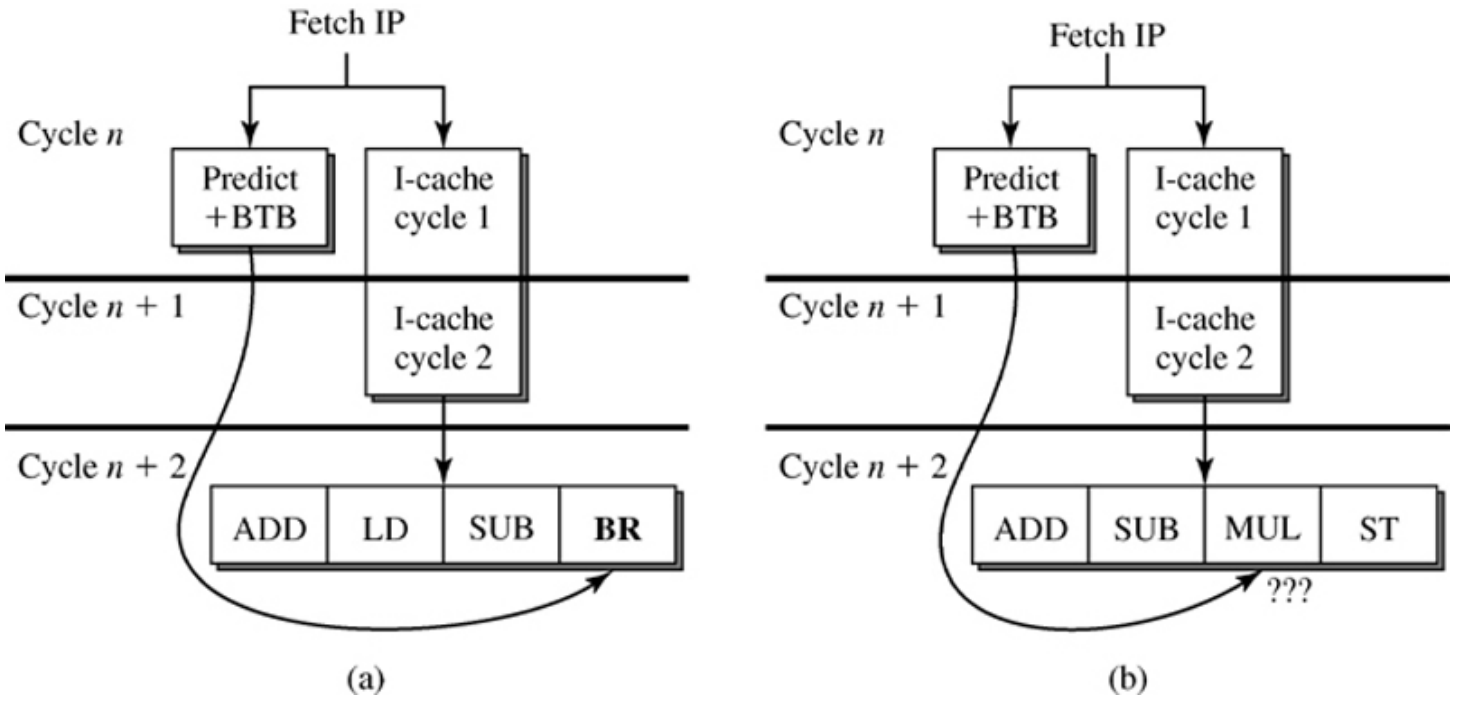
Notes:

- GHR – Global History Register, VGHR – virtual GHR
- VPCA – Virtual PC Address (distinct for different virtual branches)

- In first iteration: VPCA = PC; and VGHR = GHR; next VPCA ← Hash(PC, iter); VGHR ← Left-Shift(VGHR); repeat to find taken prediction or MAX_ITER

Phantom branch / bogus branch

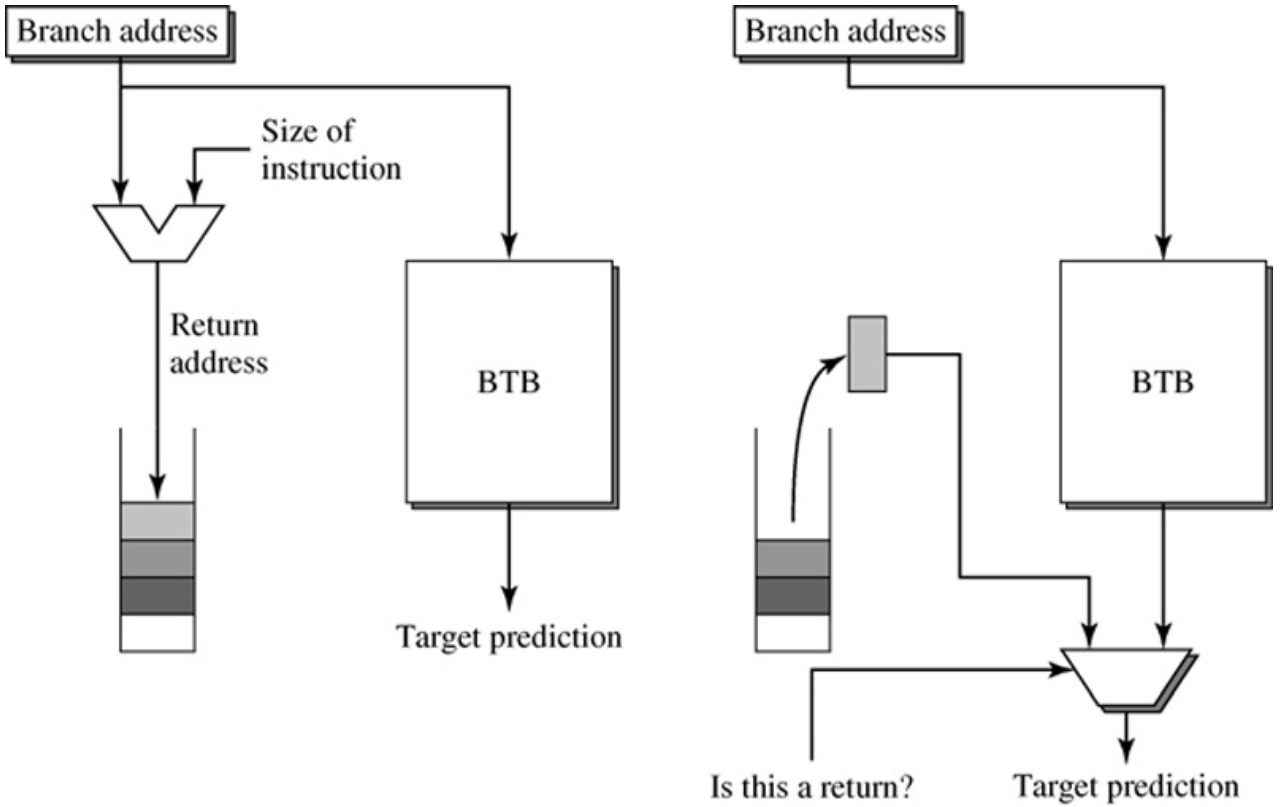
What cause these bogus predictions?



Only the part of the instruction address is used as a tag and its compare to reduce HW resources, but then BTB entry can be selected for incorrect address. If there is no branch instruction in the fetch block, then it is found 2 cycles later after fetch and instead of predicted address next fetch block is loaded. But this optimization is source of Spectre and Meltdow security vulnerabilities. Full PC address or some randomly selected hash have to be used instead.

Branch **target** speculation

- **RAS predictor** - return address stack – specialized predictor
- For the prediction of return address (MIPS jr $ra, x86 ret, etc) – HW stack. The stack depth of 8 gives >97% of prediction success
  (MIPS R10000 stack depth: 1, Alpha 21164 stack depth: 12, Intel Pentium III stack depth: 16)



Speculation may be wrong!!!

Modern processors use:

- Control speculation (conditional branches)
- Data speculation (Load/Store speculation)
  - The load is executed before the store address of previous store instruction is computed (e.g. Itanium, Power 5, Core 2)
- Sequential semantics of executed program have to be satisfied
- It means: execution of program satisfies
  - Condition #1 – the processor produces exactly the same results as the processor executing the program strictly instruction-by-instruction in program order
  - Condition #2 – the processor generates the same exceptions as…
- Next sufficient conditions exist to warrant this goal:

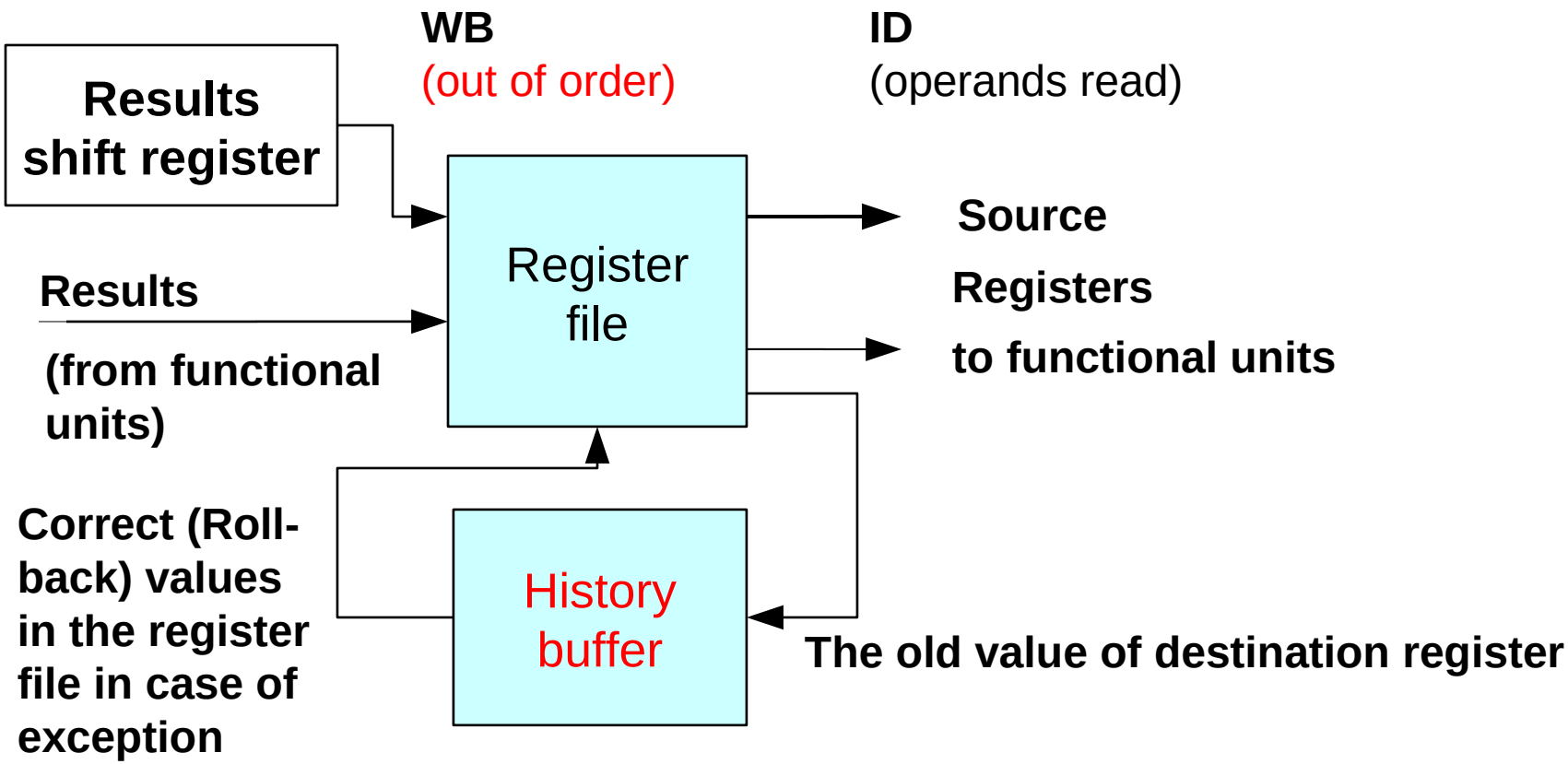
Sufficient conditions

- Cond. A – respect the data dependencies
  - Instructions wait for resolving data dependencies
- Cond. B – respect the control dependencies
  - Do not branch until the branch address is known and the branch condition is computed (and satisfied)
- Cond. C – Precise exception for interrupts and exceptions
- During the speculation, the conditions A and B are not partially or temporarily fulfilled, but the result after retire phase has to be correct
- The program execution has to be correct, and Condition C have to be guaranteed in any case

Wrong Speculation?

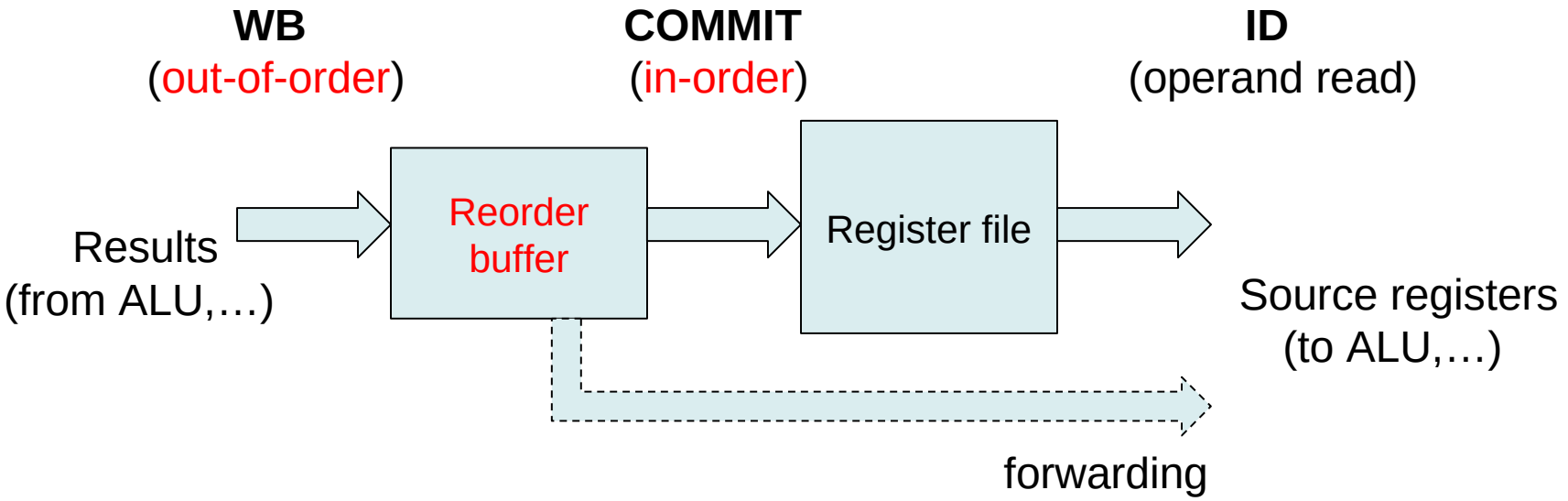
- Misprediction recovery/restart has to happen. Both options are expensive, even ten or more cycles. Two options with additional queues
  - **History Buffer** HB - M88110,…
  - **Reorder Buffer** RB – the most of today architectures. Our focus…

History Buffer



Reorder Buffer

- Results are stored in the Reorder buffer during Write-Back
- Commit: Complete the instruction in-order (Results are stored in the Register file only when all preceding instructions are complete
- Note: Holding the renamed values in Reorder buffer (ROB) is not the only possibility. Recall that it exists: Stand-alone rename register file and Merged register file => in these cases, ROB is also used (in-order complete)

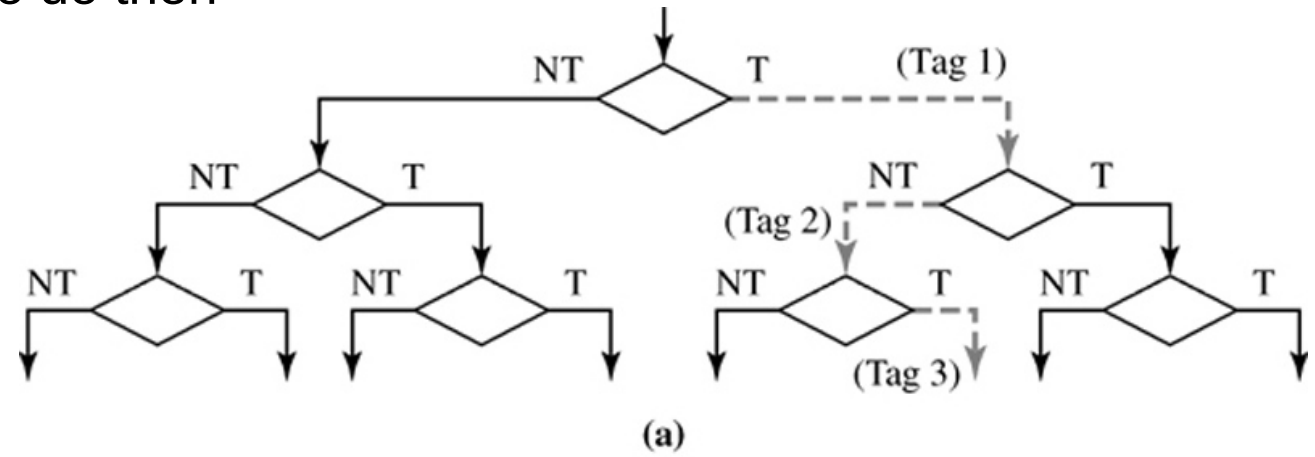


History Buffer vs Reorder Buffer

- Advantages of HB
  - Values from HB is not necessary to forward (no associative search in HB) ⇒ simpler forwarding
  - HB access not on the critical path
- Disadvantages of HB
  - Need to unwind the history buffer upon an exception (increased latency to handle the exception)
  - Additional read port for register file to read previous value of destination register (problem to scale into superscalar)
- Throughput (for execution) is similar for both approaches. Some additional latency for the exception for HB. Problem with STORE instruction.

Branch validation/recovery

- Branch speculation – frond-end stages of the pipeline – discussed till now
- Branch validation/recovery – when the direction is finally resolved?
- What to do then

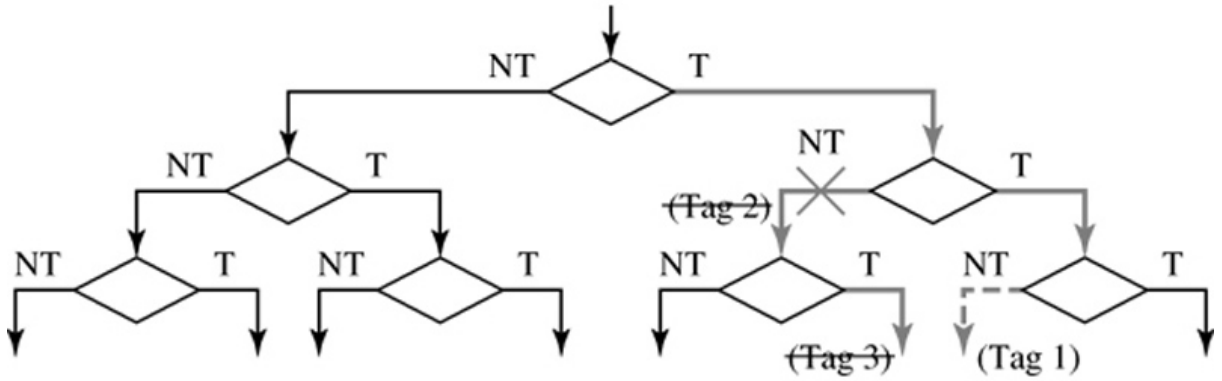


(a)

- During Branch speculation, all instructions have to be identified by the unique Tag related to certain speculative block. These tagged instructions keep information if they are speculative and to which speculative block belongs.
- During Branch speculation addresses of all branch instructions are stored (in the buffer) – it is required for Branch recovery

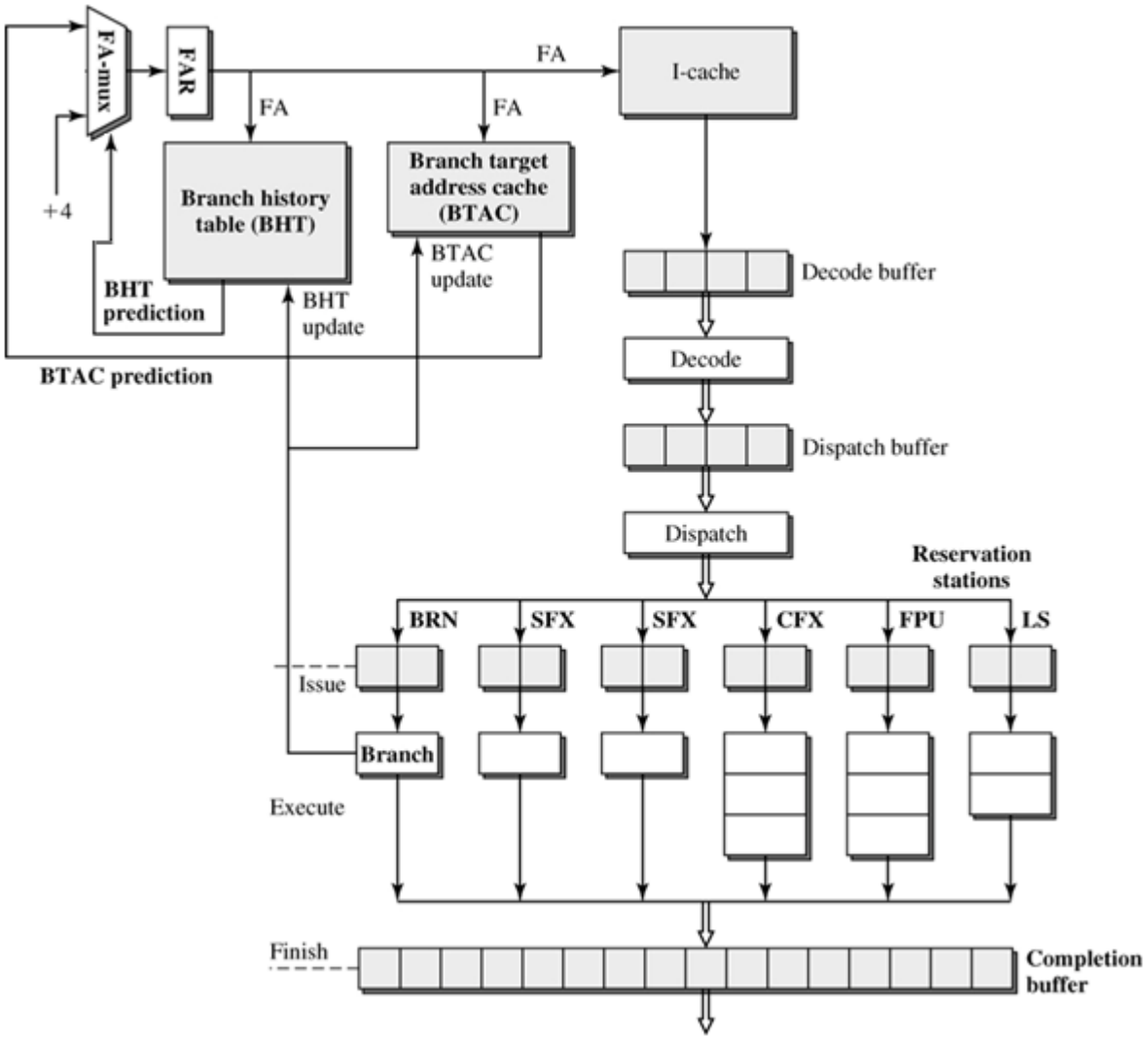
Branch validation/recovery

- Branch validation/recovery – when the actual direction of a branch is resolved



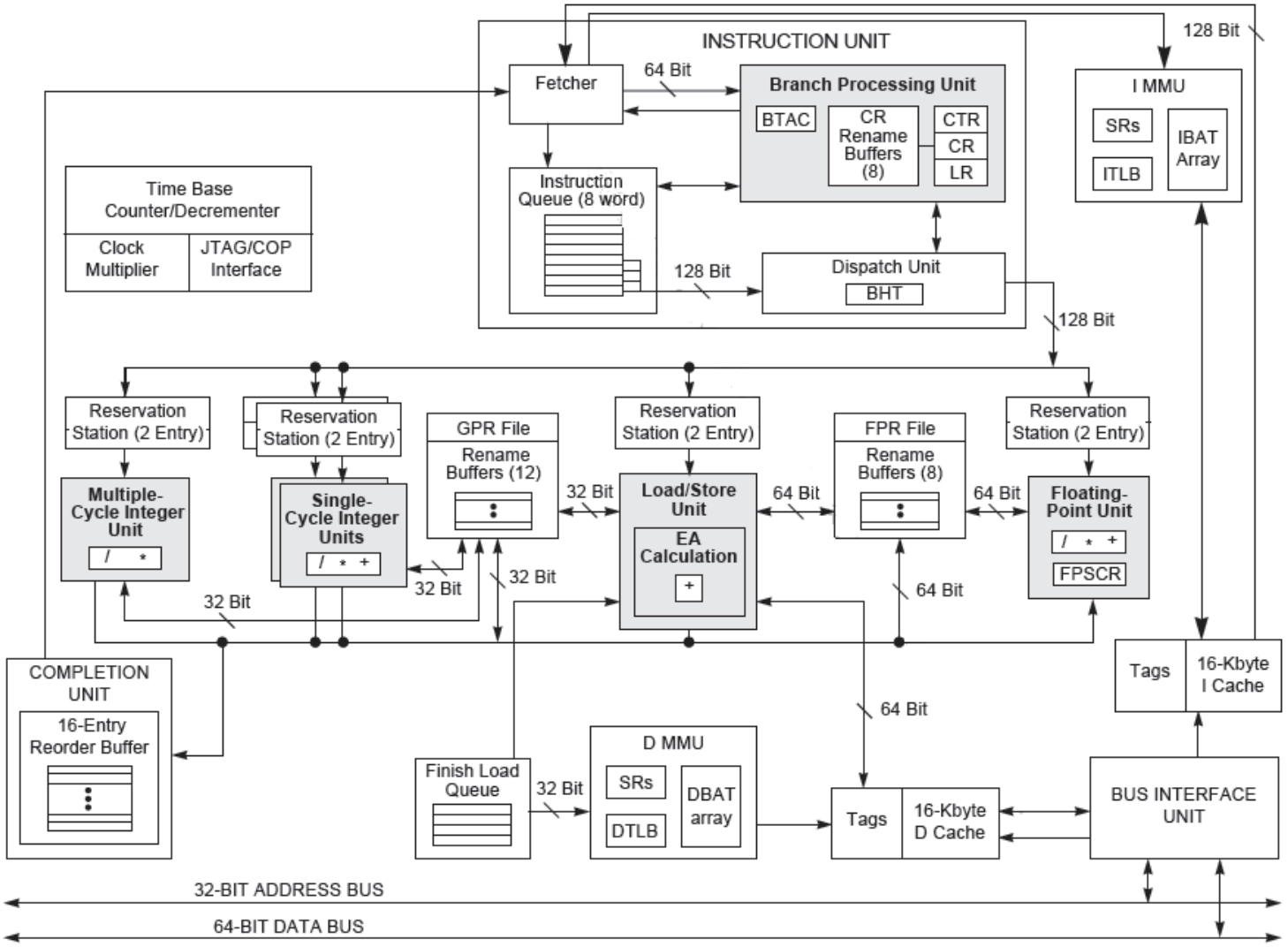
- **Correct speculation (prediction)**:
  speculation tag is deallocated, instructions become nonspeculative (allowed to complete)
- **Incorrect speculation (prediction)** :
  Two actions: the incorrect path must be terminated and the correct path must be initiated.
  The termination of the incorrect path: Discarding of all instructions (by using speculation Tags) in decode and dispatch buffers, in reservation stations, etc.
  The initiation of the correct path: Update of PC with a new address (computed branch target address or buffered sequential address during Branch speculation)

Branch prediction - PowerPC 604



- 4-wide superscalar processor (4 instructions can be decoded or completed in one cycle)
- Execute 6 instructions/cycle
- BTAC – Branch Target address cache (fully associative, 64 entries)
- BHT – Branch History Table (direct mapped, 512 entries)
- BTAC – one cycle, BHT – two cycles
- Organization FA-mux is more complex than in the picture

PowerPC 604 – different view



Instruction prefetching…

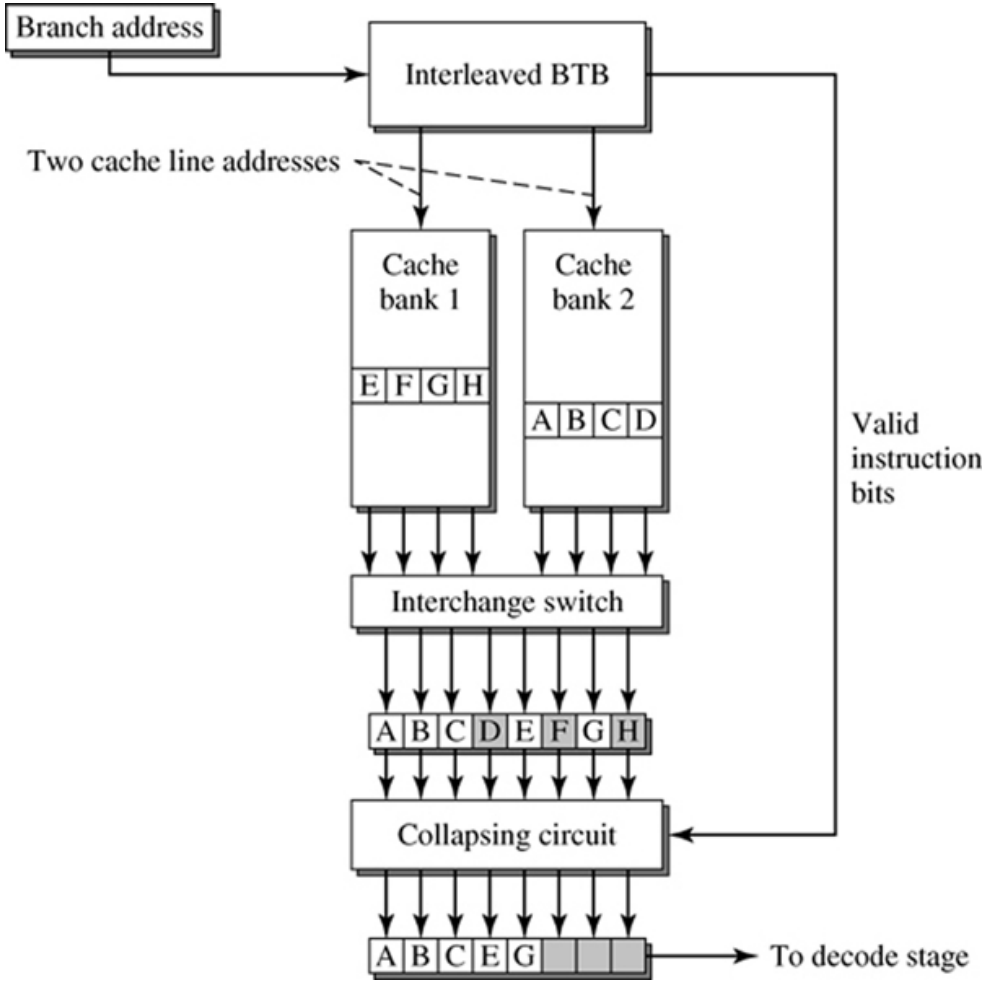
- For the superscalar processor, it is very important to predict the direction and the target of the branch correctly!
- In one cycle, it may be required to process several branch instructions (depending on the fetch group size). For example, for fetch group size 4 instructions, all can be branches. The optimal solution is to use all branch instruction addresses. Only the first one is often used.
- Another mechanism is a **predication** (or eager execution) – instructions from both paths can be executed. The instruction will only be completed if the predicate is true.
  - It ensures that CPU use cycles to process instructions from right target, but ensures that time to process another is wasted
  - Not useful for highly predictable branches
  - Power consumption

High-Bandwidth Fetch Mechanism

- Assuming a limit of one i-cache access per cycle, the number of instructions fetched from the i-cache is bounded by the line size.
- Typically, the fetch group contains branch instruction (for typical applications, 20% are branches). Usual distance to the branch target is 5 to 6 instructions for typical applications.
- Only if the taken branch is the last instruction in the fetch group, the whole fetch group contain useful instructions.. Otherwise, the fetch group (or i-cache) do not provide N instructions (where N is the fetch group width). What is the consequence?
- A similar problem is a situation where a block of instructions is split across cache lines.
- The solution is:
  - **Collapsing buffer** – alignment of instructions from non-sequential addresses
  - **Trace cache –** which stores the actual sequence of instructions following given branch – these are (speculatively) executed instead of cache access

High-Bandwidth Fetch Mechanism

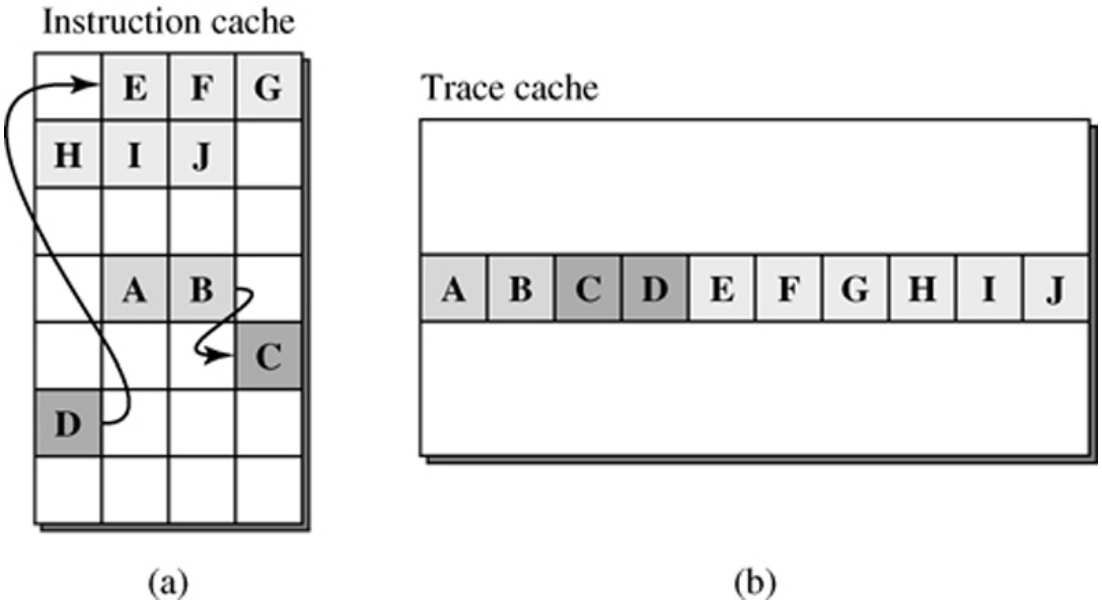
- **Collapsing buffer** – aligns not consecutive instruction into block



- It requires a banked instruction cache – more than one cache line accessible in parallel
- and interleaved BTB.
- Suppose that we want to execute the sequence of instructions:
  A, B, C, E, G – i.e. C and E are branches, E and G are their targets
- Conventional cache:
  - 1.cycle: A,B,C,D (three useful inst.)
  - 2.cycle: E,F,G,H (two useful inst.)
- BTB has to provide the information (valid instruction bits) that specify which instructions in the cache line are the part of the predicted path
- **It is hard to scale the Collapsing buffer over more than 2 cache lines / cycle**

High-Bandwidth Fetch Mechanism

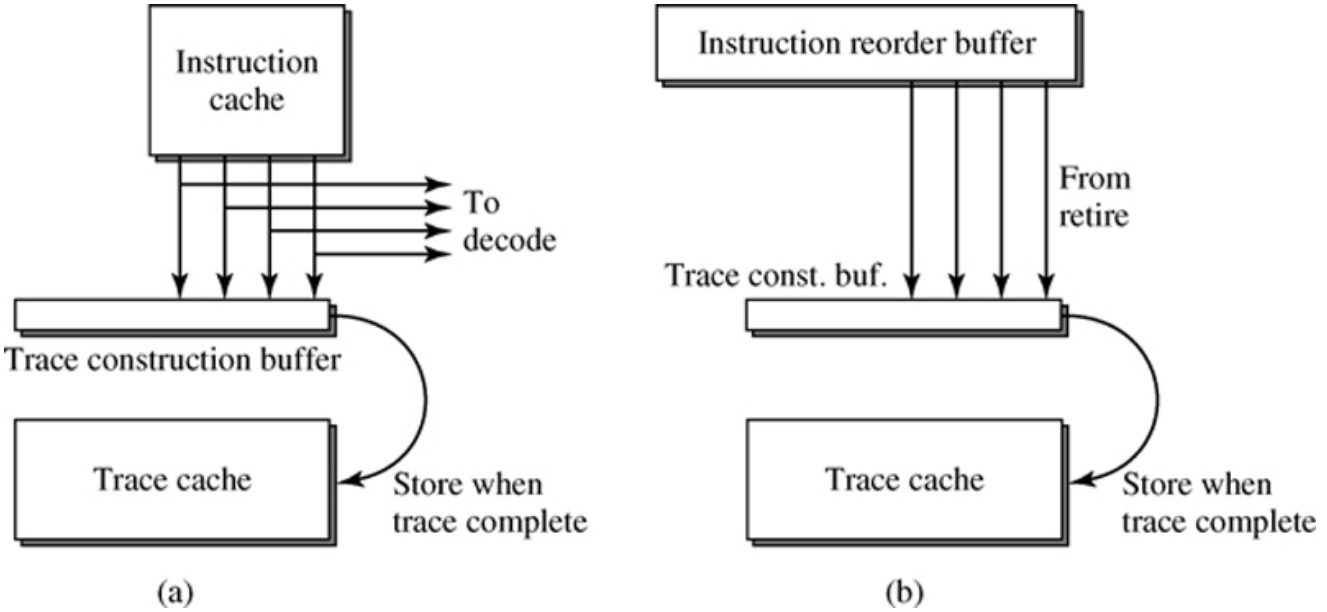
- **Trace cache** – it traces sequence of the following executed instructions for given branch/jump – in the case of trace cache hit we do not fetch from the instruction cache…
- The idea is depicted in the picture:



- The task to solve – how to fill this trace cache ???

High-Bandwidth Fetch Mechanism – Trace cache

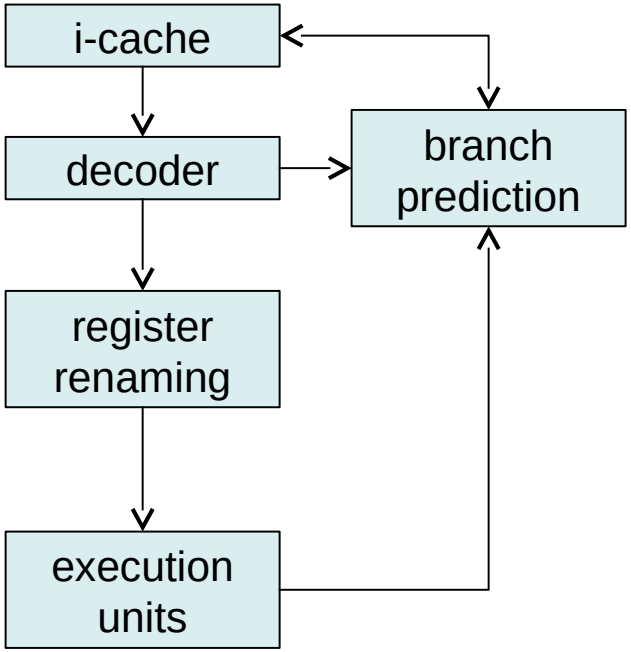
- Two possibilities from where to include fill of *Trace cache*: (a) and (b)
- When a Trace construction buffer is full (it may be determined not only by the number of instructions in the buffer, but also by the number of branches – mainly in the case (a)), the trace is stored in the trace cache

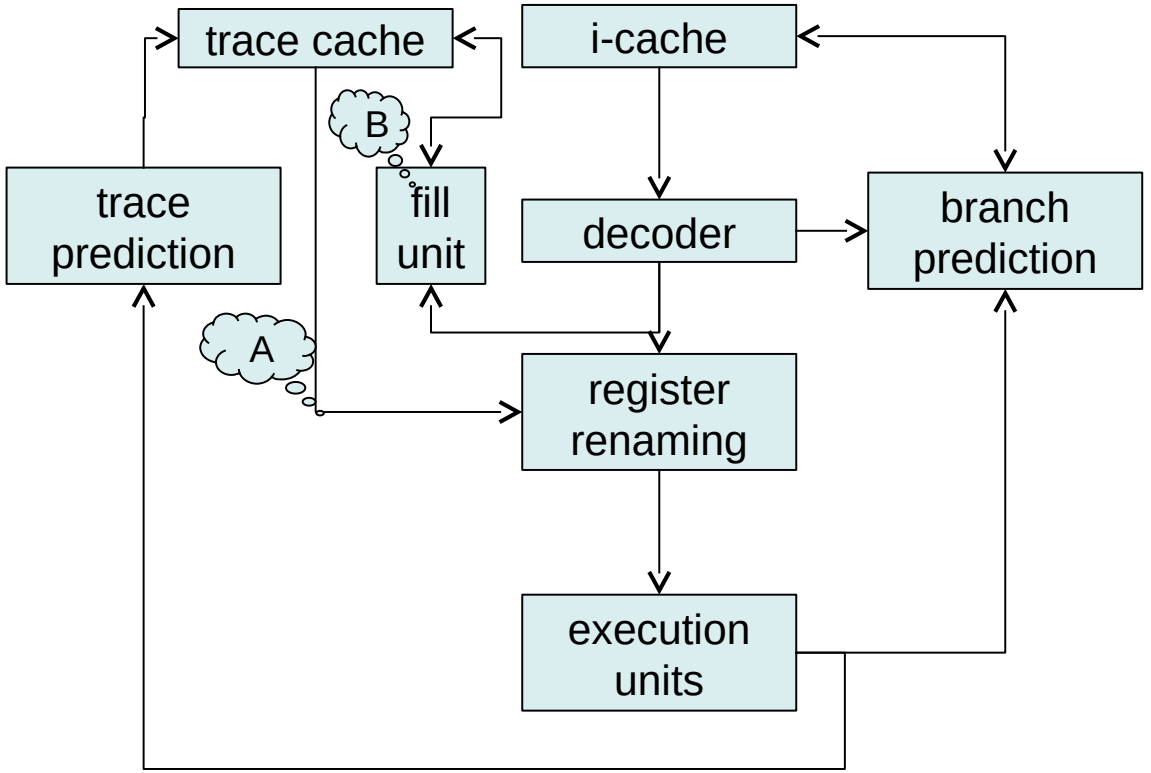


- Every trace in trace cache has to provide the starting addresses of individual blocks included in the trace. The fetch unit has to provide starting addresses of the predicted path. If all addresses match, then there is a trace cache hit. In the case of the partial match, the only subset of the trace is provided.

Trace cache

Conventional Instruction Fetch Unit without Trace Cache:





- A: entries in the trace cache can be already decoded, content can be fed directly into register renaming stage in such case ..
- B: there is one another possibility how to fill the Trace cache – from reorder buffer – after Commit as shown on previous slide

Trace cache - Pentium 4 processor, NetBurst microarchitecture

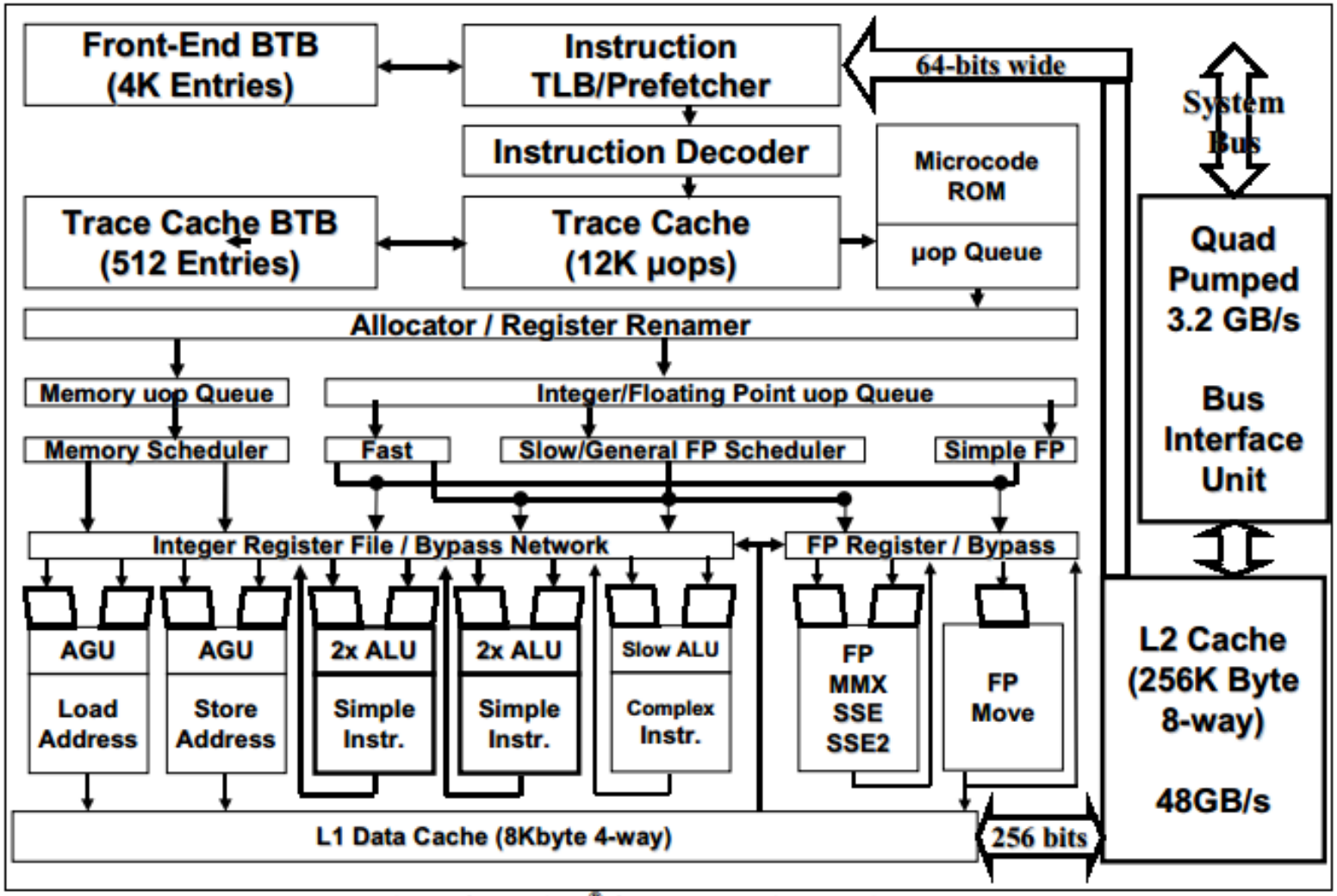


Figure 4: Pentium® 4 processor microarchitecture

Instead of conclusion benchmark the code

- Try to measure the time:

```
for (int i = 0; i < max; i++)
        if (cond) /* use -O0 else compiler optimizes the loop */
                sum++;
```

cond	pattern	time
(i & 0×80000000) == 0	always T	
(i & 0xffffffff) == 0	always F	
(i & 1) == 0	TF	
(i & 3) == 0	TFFF	
(i & 2) == 0	TTFF	
(i & 4) == 0	TTTTFFFF	
(i & 8) == 0	8T 8F	
(i & 32) == 0	32T 32F	

Instead of conclusion …

- Try to measure the time :

```
for (int i = 0; i < max; i++){
  a=0;
  if(cond n.1)   a=3;
  if((i & 2) == 0)   b=10;
  if(a <= 0)  sum++;   //if(cond n.1)  sum++;
}
```

Cond. n.1	pattern	Time A	Time B
(i & 0×80000000) == 0	always T		
(i & 0xffffffff) == 0	always F		
(i & 1) == 0	TF		
(i & 3) == 0	TFFF		
(i & 4) == 0	4T 4F		

Typical values for today CPUs

	AMD FX-8150	Intel i7 2600
Instruction Decode Width	4-wide	4-wide
Single Core Peak Decode	4 instructions	4 instructions
Instruction Decode Queue	16 entry	18+ entry
Buffers	40-entry load queue 24-entry store queue	48 load and 32 store buffers
pipeline depth	18+ stages	14 stages
branch misprediction penalty	20 clock cycles for conditional and indirect branches 15 clock cycles for unconditional jumps and returns	17 cycles
reservation station	40-entry unified integer, memory scheduler; 60-entry unified floating point scheduler	36-entry centralized reservation station shared by 6 functional unit
reorder buffer	128-entry retirement queue	128-entry reorder buffer

References:

- PowerPC604 RISC Microprocessor Technical Summary: http://www.freescale.com/files/32bit/doc/data_sheet/MPC604.pdf
- Karel Driesen: Accurate Indirect Branch Prediction http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.4874&rep=rep1&type=pdf
- Hyesoon Kim et al.: VPC Prediction: Reducing the Cost of Indirect Branches via Hardware-Based Dynamic Devirtualization http://users.ece.cmu.edu/~omutlu/pub/kim_isca07.pdf
- TseYu Yeh and Yale N Patt: Alternative Implementations of Two-Level Adaptive Branch Prediction http://www.eecg.toronto.edu/~moshovos/ACA05/read/isca-92.2-level-adaptive.pdf
- **Shen, J.P., Lipasti, M.H.: Modern Processor Design : Fundamentals of Superscalar Processors, First Edition, New York, McGraw-Hill Inc., 2005**
- **Glenn Hinton et al.: The Microarchitecture of the Pentium 4 Processor. http://www.ecs.umass.edu/ece/koren/ece568/papers/Pentium4.pdf**