CSE 520
Computer Architecture II

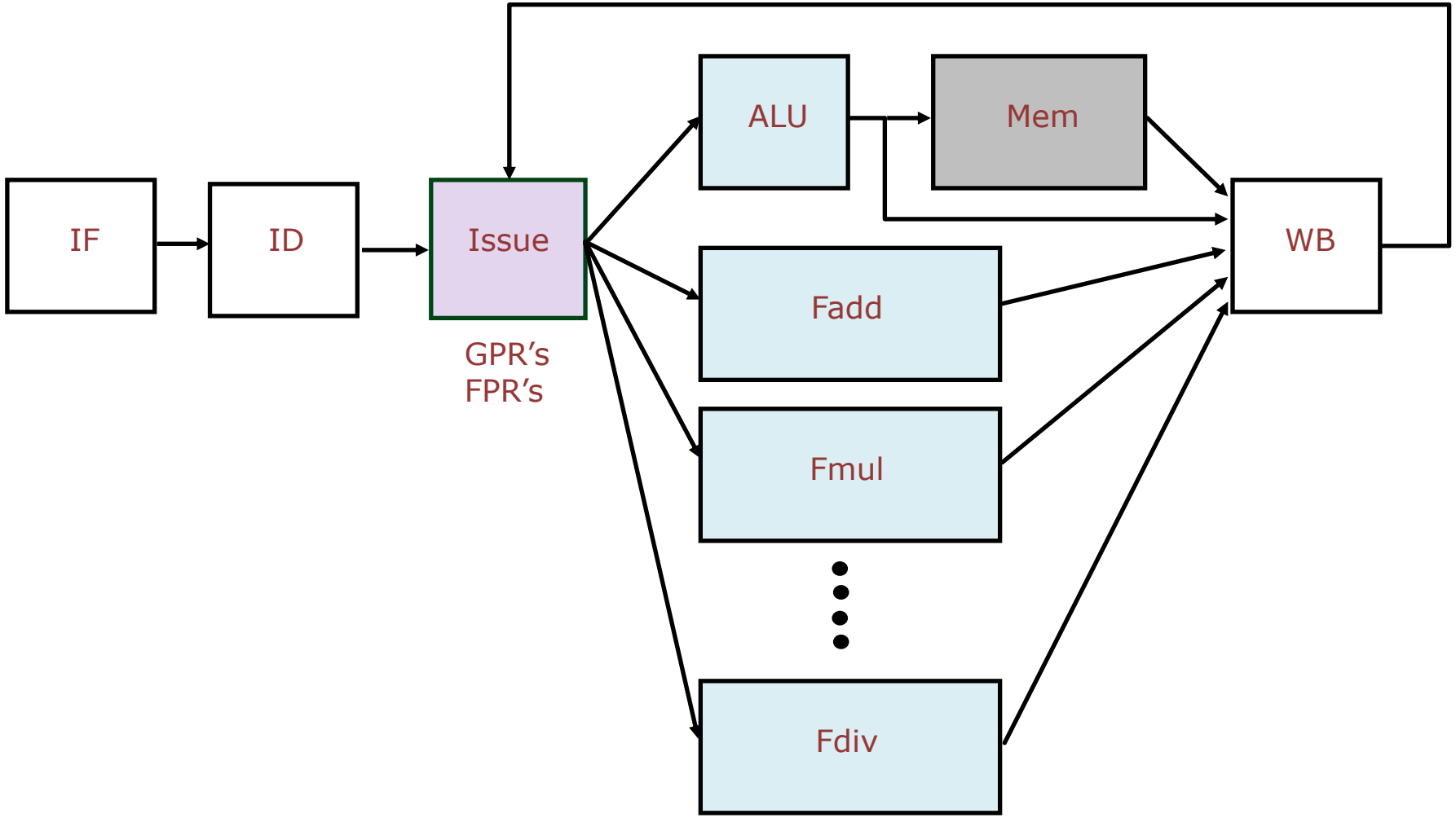
Branch Prediction

Prof. Michel A. Kinsy

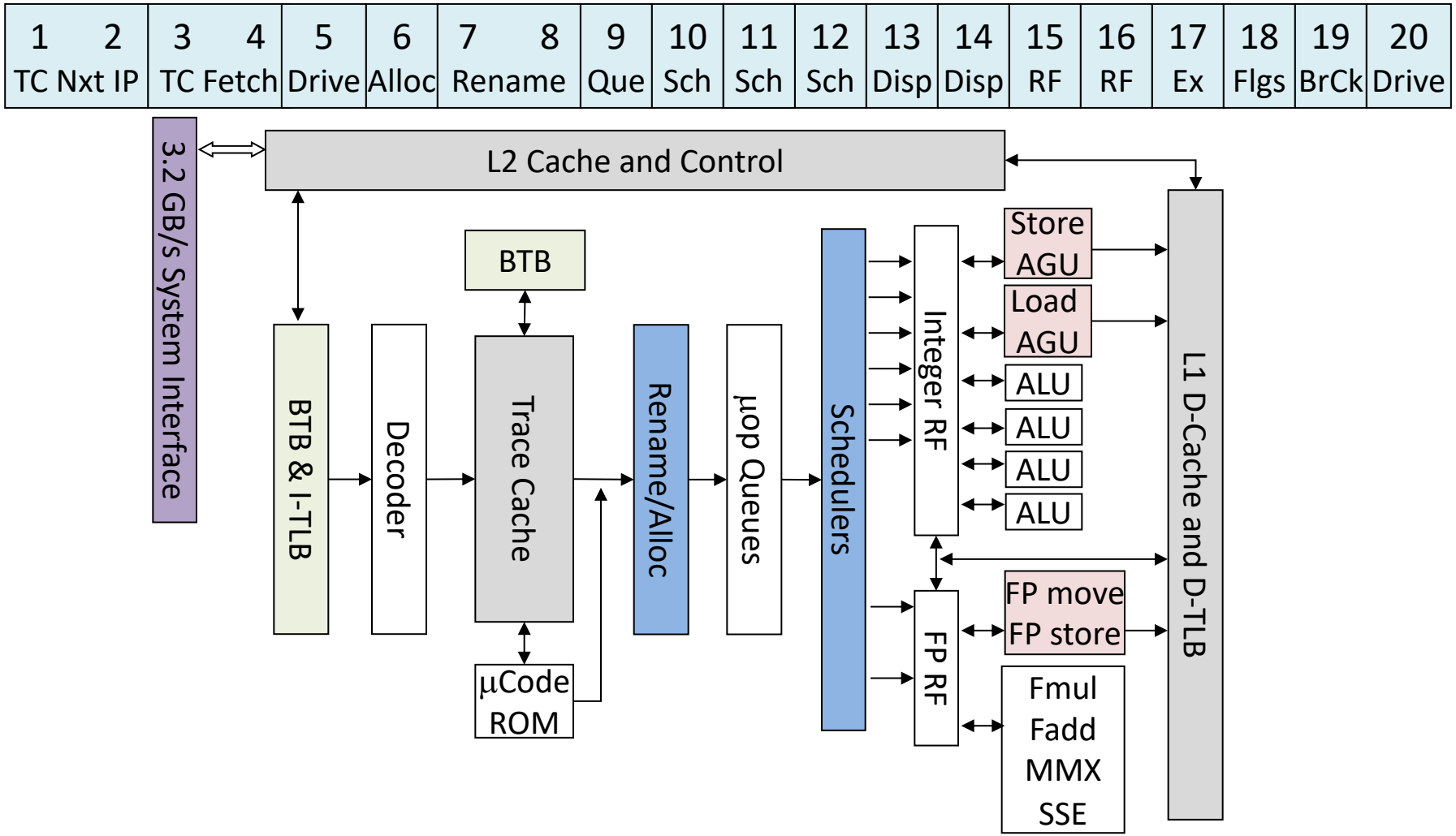
Instruction Interactions

- An instruction in the pipeline may need a resource being used by another instruction in the pipeline
  - Structural hazard
- An instruction may depend on something produced by an earlier instruction
  - Dependence may be for a data calculation
    - Data hazard
  - Dependence may be for calculating the next address
    - Control hazard (branches, interrupts)

Modern Processors: Complex Pipeline Structures

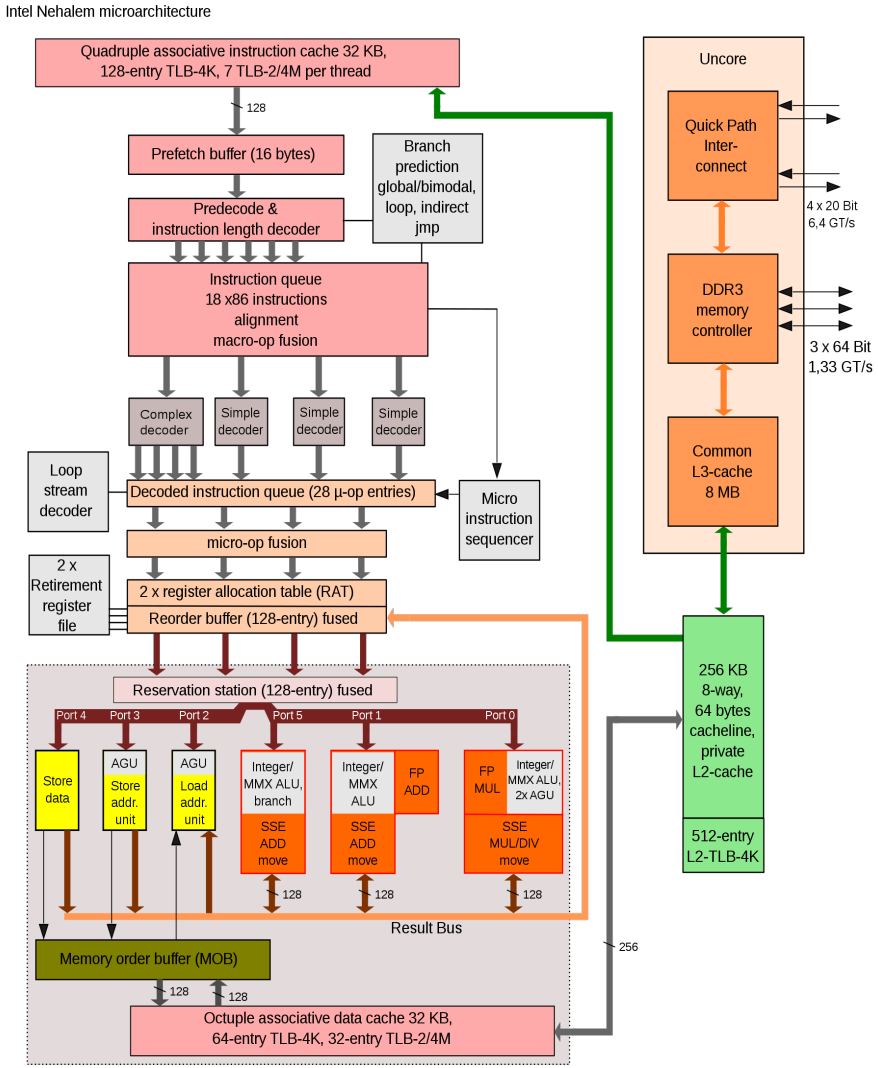


Modern Processors: Complex Pipeline Structures



Intel Pentium 4

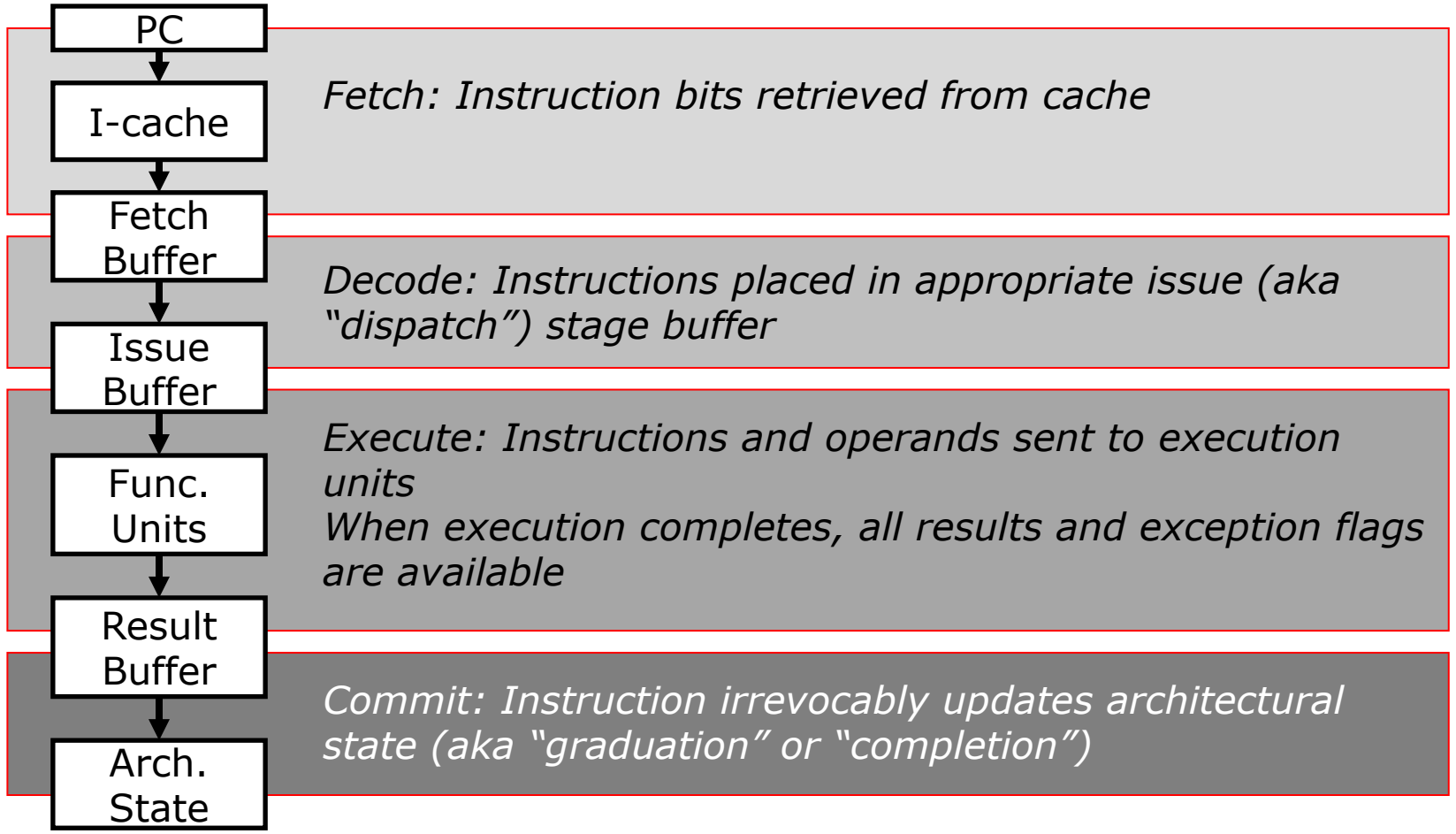
Modern Processors: Complex Pipeline Structures



Execution Concurrency Limits

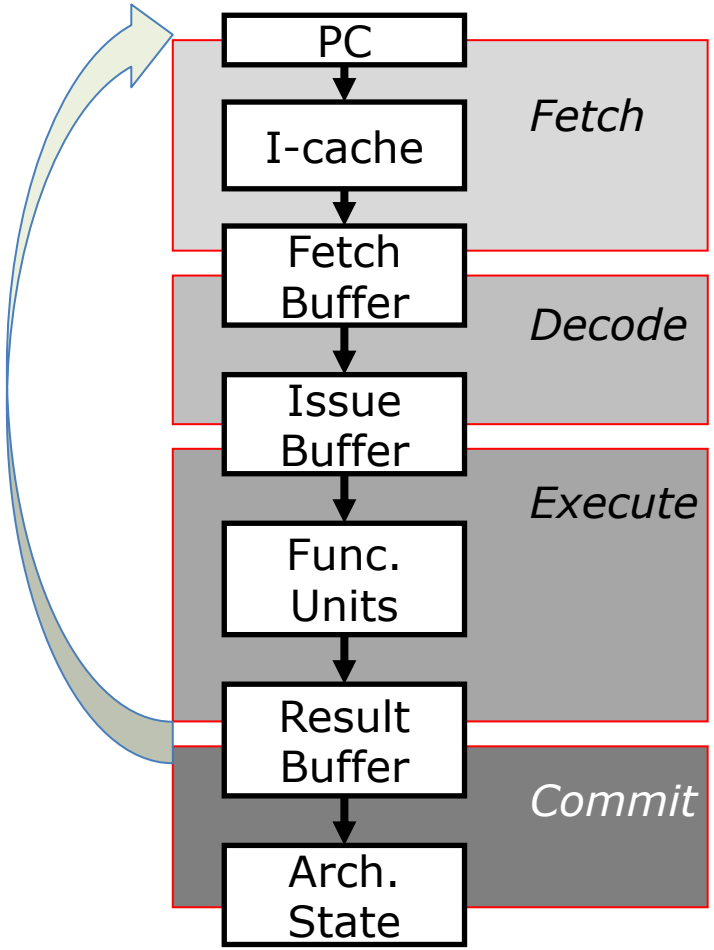
- Which features of an ISA limit the number of instructions in the pipeline?
  - Number of Registers
- Which features of a program limit the number of instructions in the pipeline?
  - Control transfers

Instruction Execution Phases



Branch Penalty

- How many instructions need to be killed on a misprediction?
  - Modern processors may have > 10 pipeline stages between next pc calculation and branch resolution!



Average Branches Distance

- Average Run-Length between Branches
  - Average dynamic instruction mix from SPEC92:

	SPECint92	SPECfp92
ALU	39 %	13 %
FPU Add		20 %
FPU Mult		13 %
load	26 %	23 %
store	9 %	9 %
branch	16 %	8 %
other	10 %	12 %

SPECint92: compress, eqntott, espresso, gcc , li
SPECfp92: doduc, ear, hydro2d, mdijdp2, su2cor

Branches and Jumps

- Each instruction fetch depends on one or two pieces of information from the preceding instruction:
  1. Is the preceding instruction a taken branch?
  2. If so, what is the target address?

Instruction	Taken known?	Target known?
J	After Inst. Decode	After Inst. Decode
JAL/JALR	After Inst. Decode	After Reg Fetch
BEQ/BNE	After Inst. Execute	After Inst. Decode

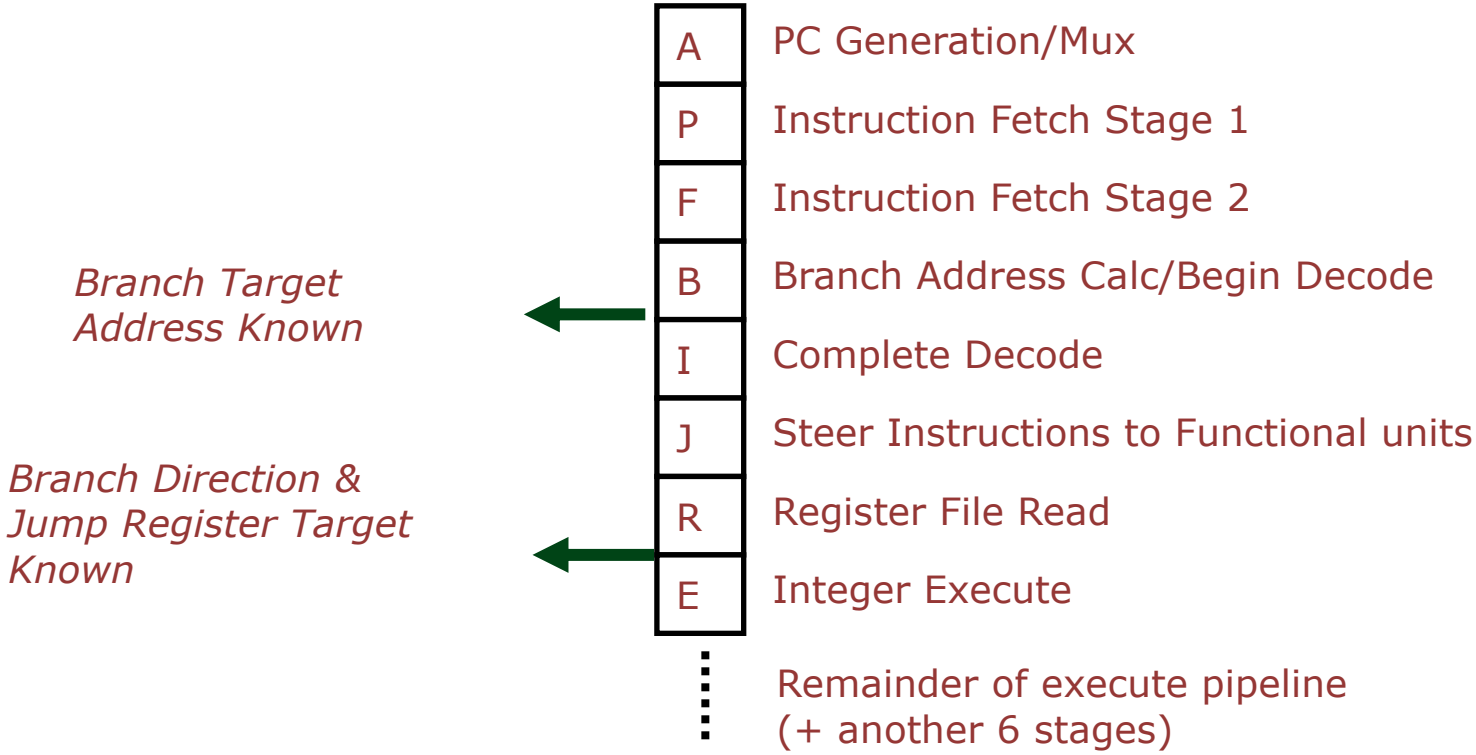
Branches and Jumps

- Each instruction fetch depends on one or two pieces of information from the preceding instruction:
  1. Is the preceding instruction a taken branch?
  2. If so, what is the target address?

Type	Direction at fetch time	Number of possible next fetch addresses?	When is next fetch address resolved?
Conditional	Unknown	2	Execution (register dependent)
Unconditional	Always taken	1	Decode (PC + offset)
Call	Always taken	1	Decode (PC + offset)
Return	Always taken	Many	Execution (register dependent)
Indirect	Always taken	Many	Execution (register dependent)

Branch Penalties in Modern Pipelines

- UltraSPARC-III instruction fetch pipeline stages (in-order issue, 4-way superscalar, 750MHz, 2000)



Control Dependencies

- Branches are very frequent
  - Approx. 20% of all instructions
- Can not wait until we know where it goes
  - Long pipelines
    - Branch outcome known after x cycles
    - No scheduling past the branch until outcome known
  - Superscalars (e.g., 4-way)
    - Branch every cycle or so!
    - One cycle of work, then bubbles for ~x cycles?

Control Flow Penalty

- Assume a 4-wide *superscalar* pipeline with 20-cycle branch resolution latency
- How long does it take to fetch 400 instructions?
  - Assume no fetch breaks and 1 out of 4 instructions is a branch
  - 100% accuracy
    - 100 cycles (all instructions fetched on the correct path)
    - No wasted work
  - 99% accuracy
    - 100 (correct path) + 20 (wrong path) = 120 cycles
    - 20% extra instructions fetched
  - 98% accuracy
    - 100 (correct path) + 20 * 2 (wrong path) = 140 cycles
    - 40% extra instructions fetched
  - 95% accuracy
    - 100 (correct path) + 20 * 5 (wrong path) = 200 cycles
    - 100% extra instructions fetched

Reducing Control Flow Penalty

- Software solutions
  - Eliminate branches - loop unrolling increases the run length
  - Reduce resolution time - instruction scheduling compute the branch condition as early as possible (of limited value)
- Hardware solutions
  - Stall the pipeline until we know the next fetch address
  - Guess the next fetch address
    - Speculate - branch prediction speculative execution of instructions beyond the branch
  - Employ delayed branching (branch delay slot)
  - Do something else (fine-grained multithreading)
  - Eliminate control-flow instructions (predicated execution)
  - Fetch from both possible paths (if you know the addresses of both possible paths)
    - Multipath execution

Branch Prediction

- Motivation:
  - Branch penalties limit performance of deeply pipelined processors
  - Modern branch predictors have high accuracy (>95%) and can reduce branch penalties significantly
- Required hardware support:
  - Prediction structures: branch history tables, branch target buffers, etc.
  - Mispredict recovery mechanisms:
    - Keep result computation separate from commit
    - Kill instructions following branch in pipeline
    - Restore state to state following branch

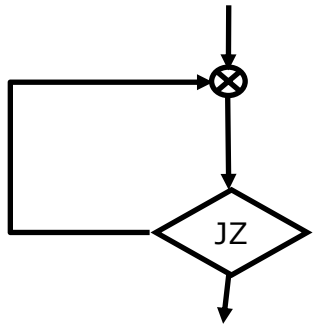
Branch Prediction Techniques

- Compile time (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)
  - Program analysis based (likely direction)
- Run time (dynamic)
  - Last time prediction (single-bit)
  - Two-bit counter based prediction
  - Two-level prediction (global vs. local)
  - Hybrid

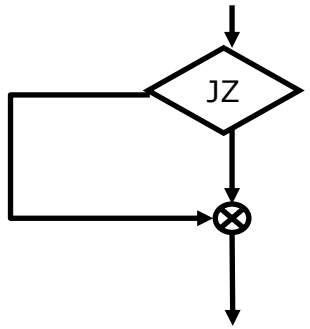
Static Branch Prediction

- Overall probability a branch is taken is ~60-70% but:

backward 90%



forward 50%



- ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110
  - bne0 (preferred taken) beq0 (not taken)
- ISA can allow arbitrary choice of statically predicted direction, e.g., HP PA-RISC, Intel IA-64
  - Typically reported as ~80% accurate

Static Branch Prediction

- Always not-taken
  - Simple to implement: no need for BTB, no direction prediction
  - Low accuracy: ~30-40%
  - Compiler can layout code such that the likely path is the "not-taken" path
- Always taken
  - No direction prediction
  - Better accuracy: ~60-70%
    - Backward branches (i.e. loop branches) are usually taken
    - Backward branch: target address lower than branch PC
- Backward taken, forward not taken (BTFN)
  - Predict backward (loop) branches as taken, others not-taken

Branch Prediction Techniques

- Compile time (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)
  - Program analysis based (likely direction)
- Run time (dynamic)
  - Last time prediction (single-bit)
  - Two-bit counter based prediction
  - Two-level prediction (global vs. local)
  - Hybrid

Dynamic Branch Prediction

- Key Idea: Predict branches based on the dynamic execution behavior of the program
  - Collected at run-time
- Advantages
  - Prediction based on history of the execution of branches
  - It can adapt to dynamic changes in branch behavior
  - No need for static profiling
- Disadvantages
  - More complex architecture (requires additional hardware

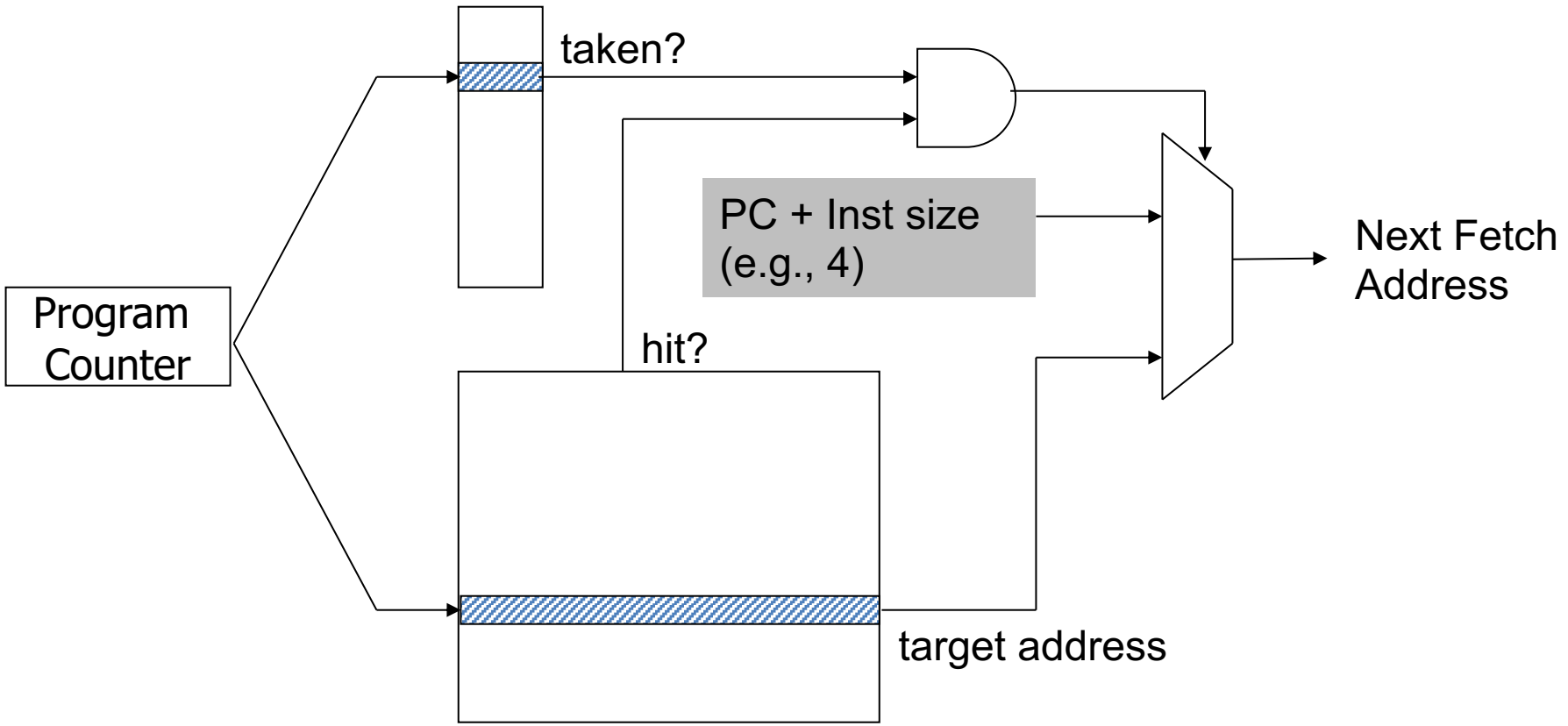
Dynamic Branch Prediction

- Learning based on past behavior
- Temporal correlation
  - The way a branch resolves may be a good predictor of the way it will resolve at the next execution
- Spatial correlation
  - Several branches may resolve in a highly correlated manner (a preferred path of execution)

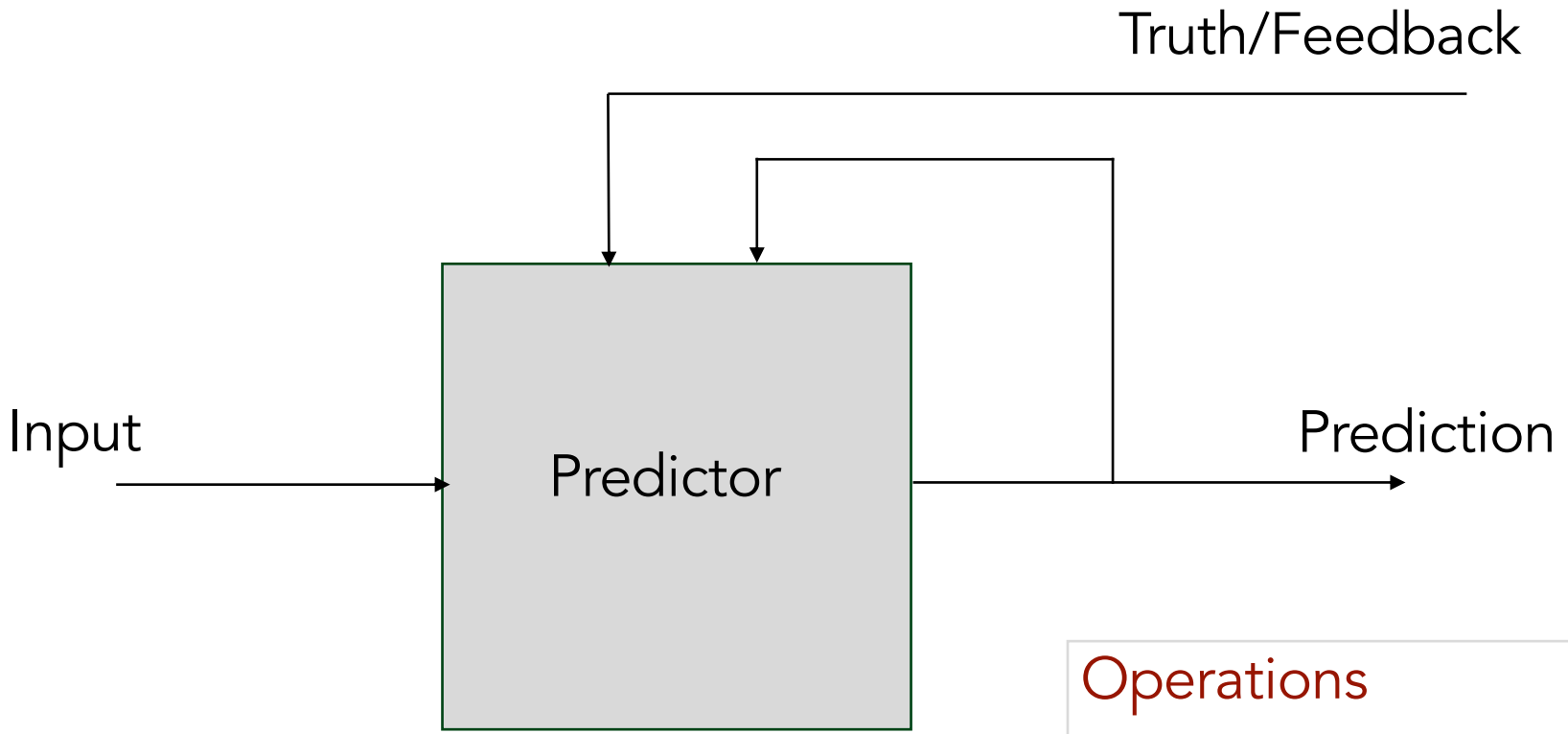
Dynamic Branch Prediction

- Requires three things to be predicted at fetch stage:
  - Whether the fetched instruction is a branch
  - (Conditional) branch direction
  - Branch target address (if taken)
    - So we also need to predict the next fetch address (to be used in the next cycle)
- Target address remains the same for a conditional direct branch across dynamic instances
  - Store the target address from previous instance and access it with the PC
  - Branch Target Buffer (BTB) or Branch Target Address Cache

Fetch Stage with Branch Target Buffer



Dynamic Prediction



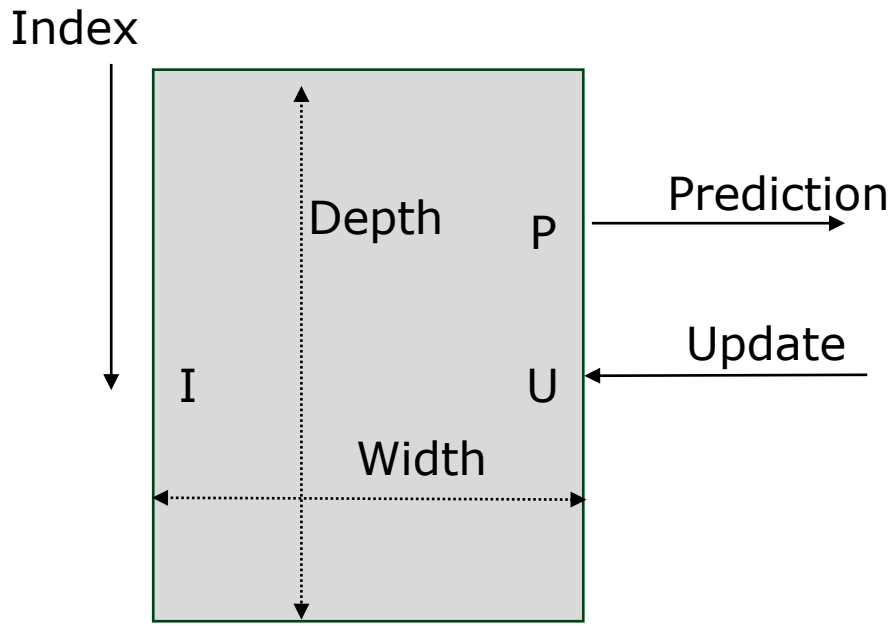
Prediction as a feedback control process

Operations

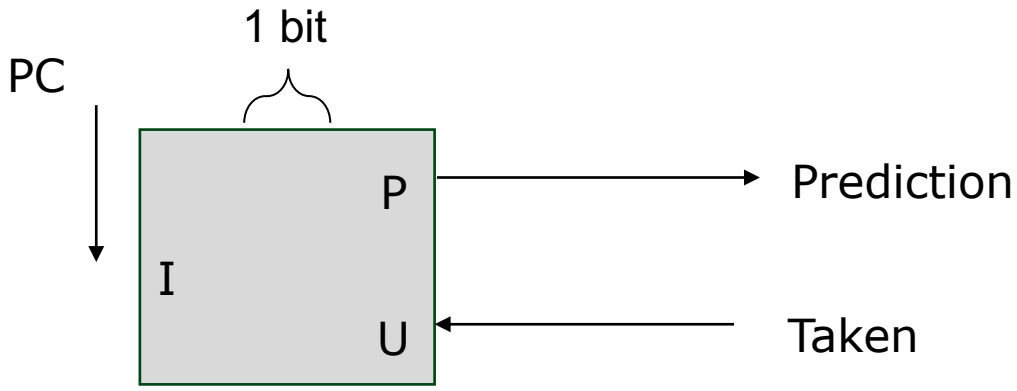
- Predict
- Update

Predictor Primitive

- Indexed table holding values
- Operations
  - Predict
  - Update
- Algebraic notation
  - Prediction = P[Width, Depth](Index; Update)



One-bit Predictor

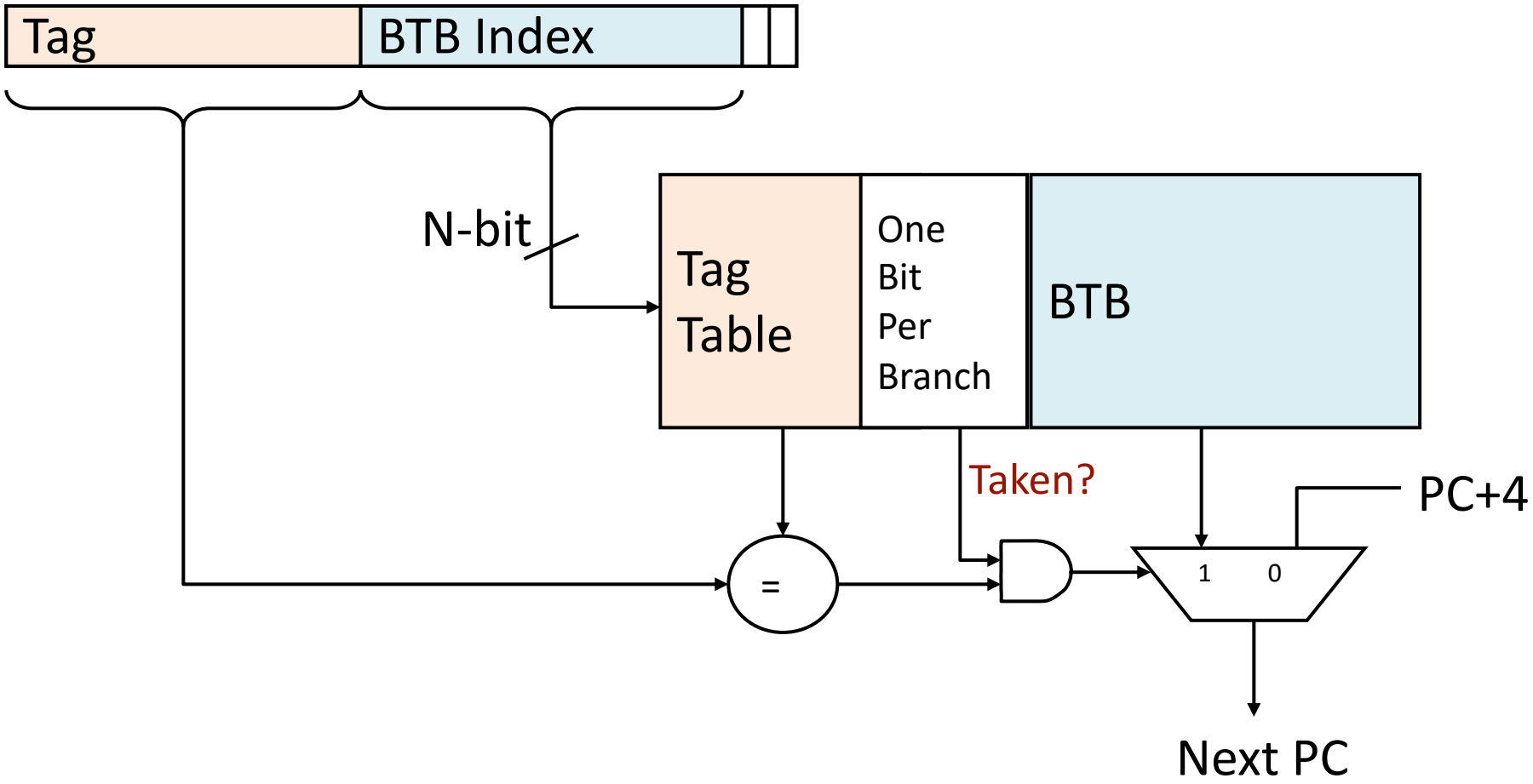


- A21064(PC; T) = P[1, 2K](PC; T)
- What happens on loop branches?
  - At best, mispredicts twice for every use of loop

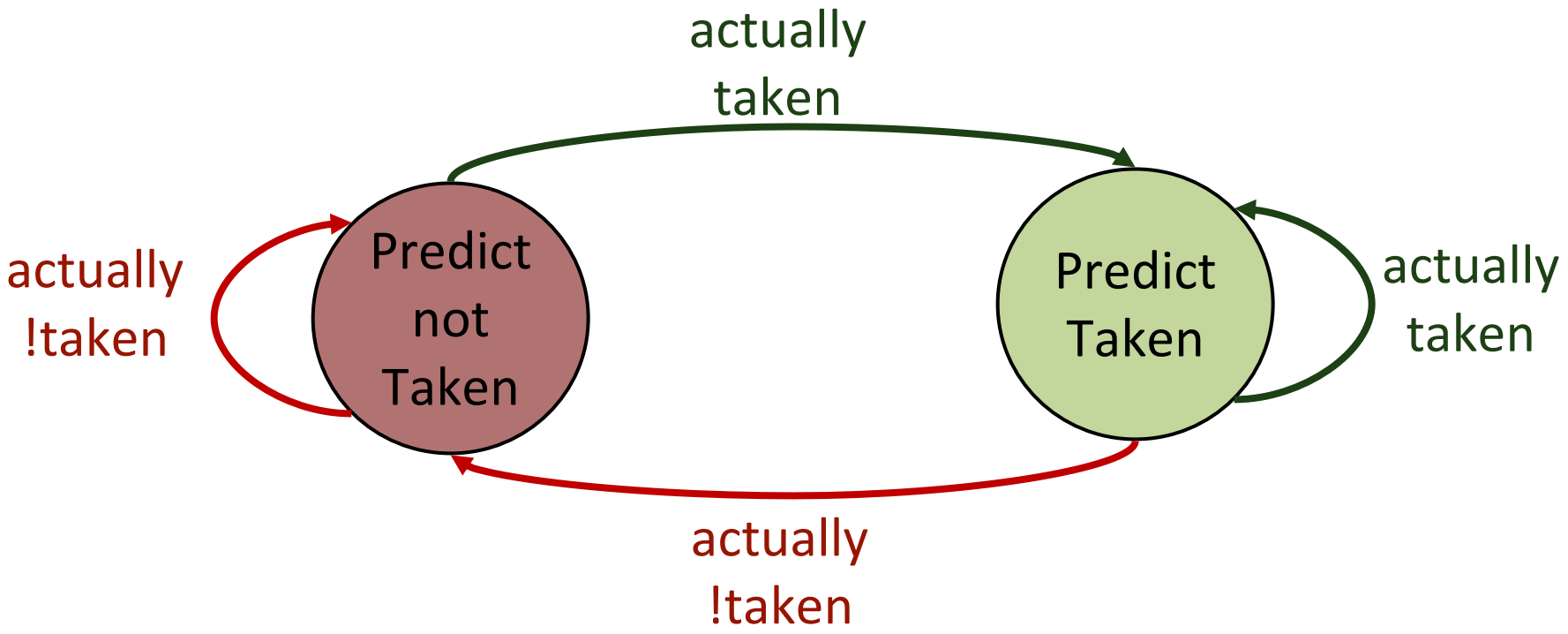
Last Time Predictor

- Last time predictor
  - Single bit per branch (stored in BTB)
  - Indicates which direction branch went last time it executed
  - TTTTTTTTTTNNNNNNNNNN → 90% accuracy
- Always mispredicts the last iteration and the first iteration of a loop branch
  - Accuracy for a loop with N iterations = (N-2)/N
  - Works well for for loops with a large number of iterations
  - Works poorly for loops with small number of iterations or non-correlated branches
  - TNTNTNTNTNTNTNTNTNTN → 0% accuracy

Implementation of the 1-bit Predictor

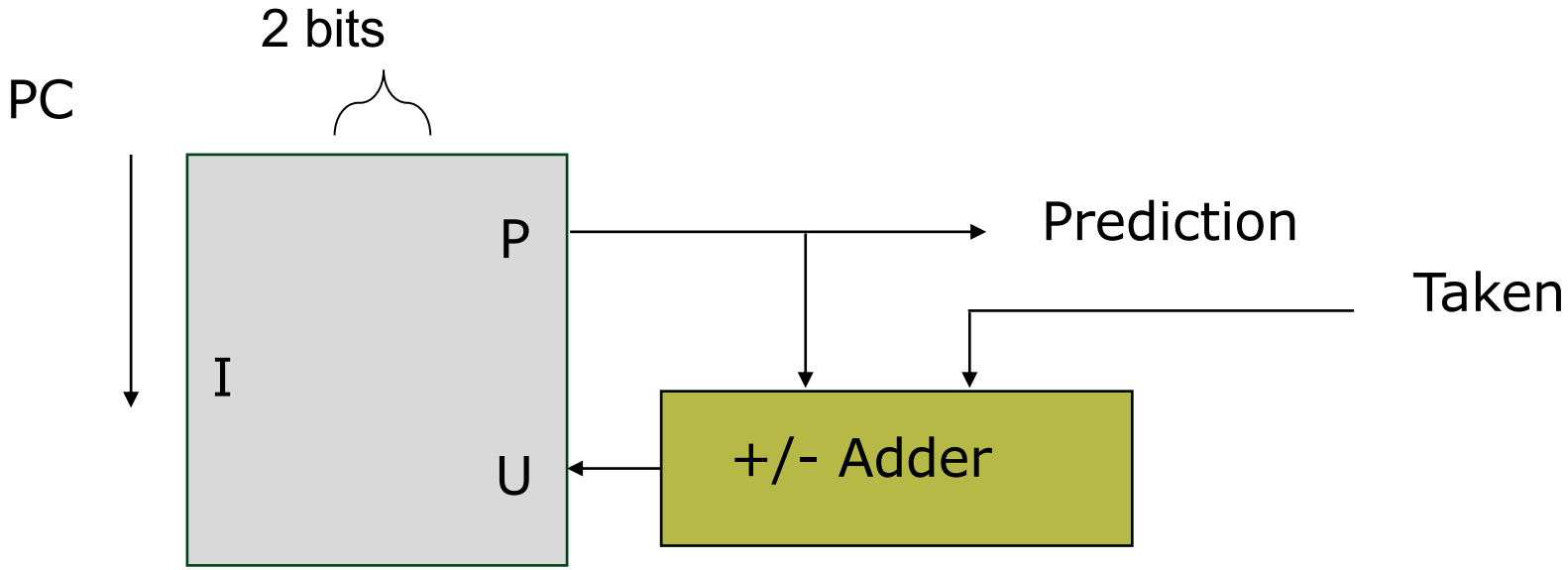


State Machine of the 1-bit Predictor



Two-bit Predictor

- Counter[W,D](I; T) = P[W, D]
  - (I; if T then P+1 else P-1)
- A21164(PC; T) = MSB(Counter[2, 2K](PC; T))

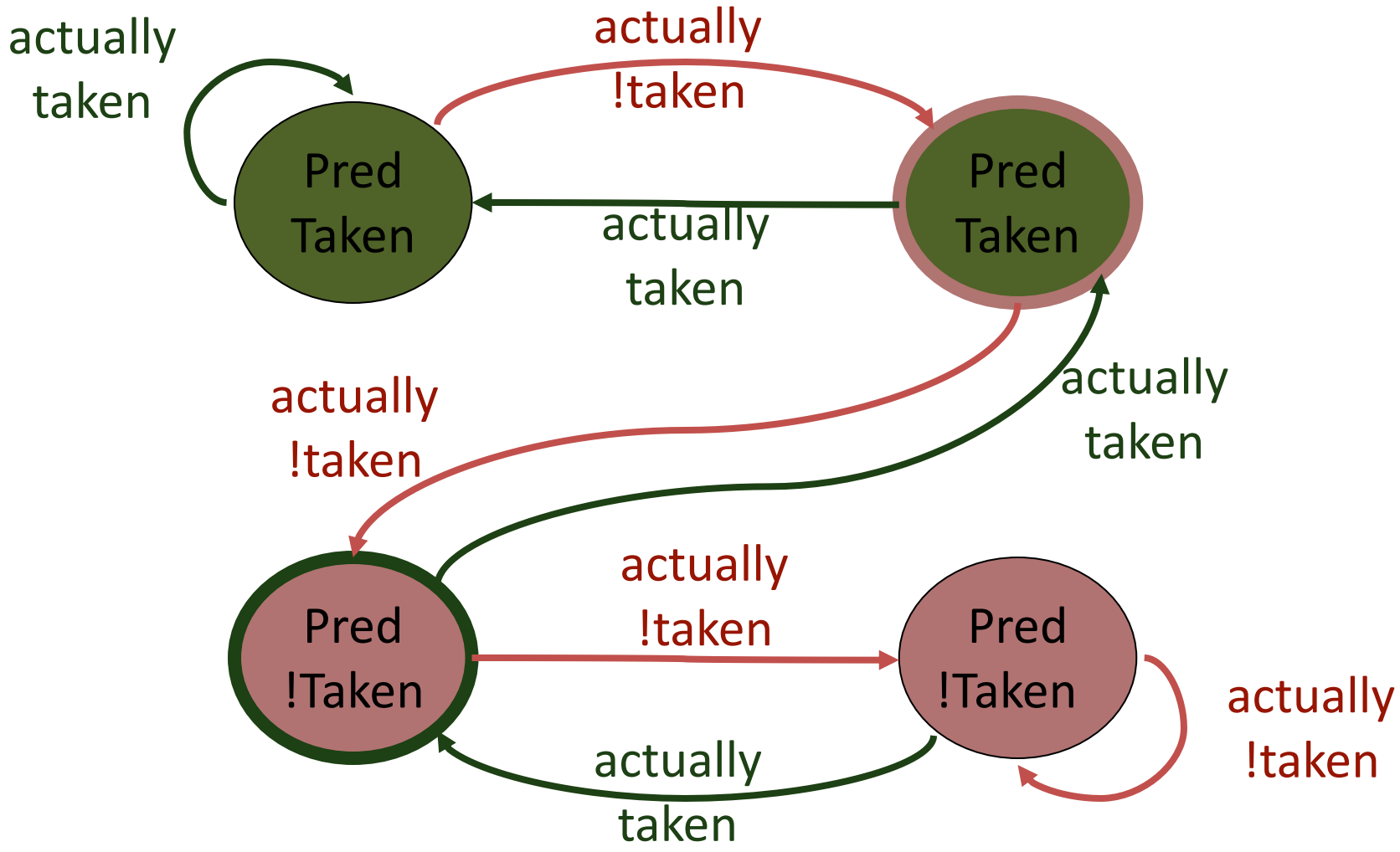


Branch Prediction Bits

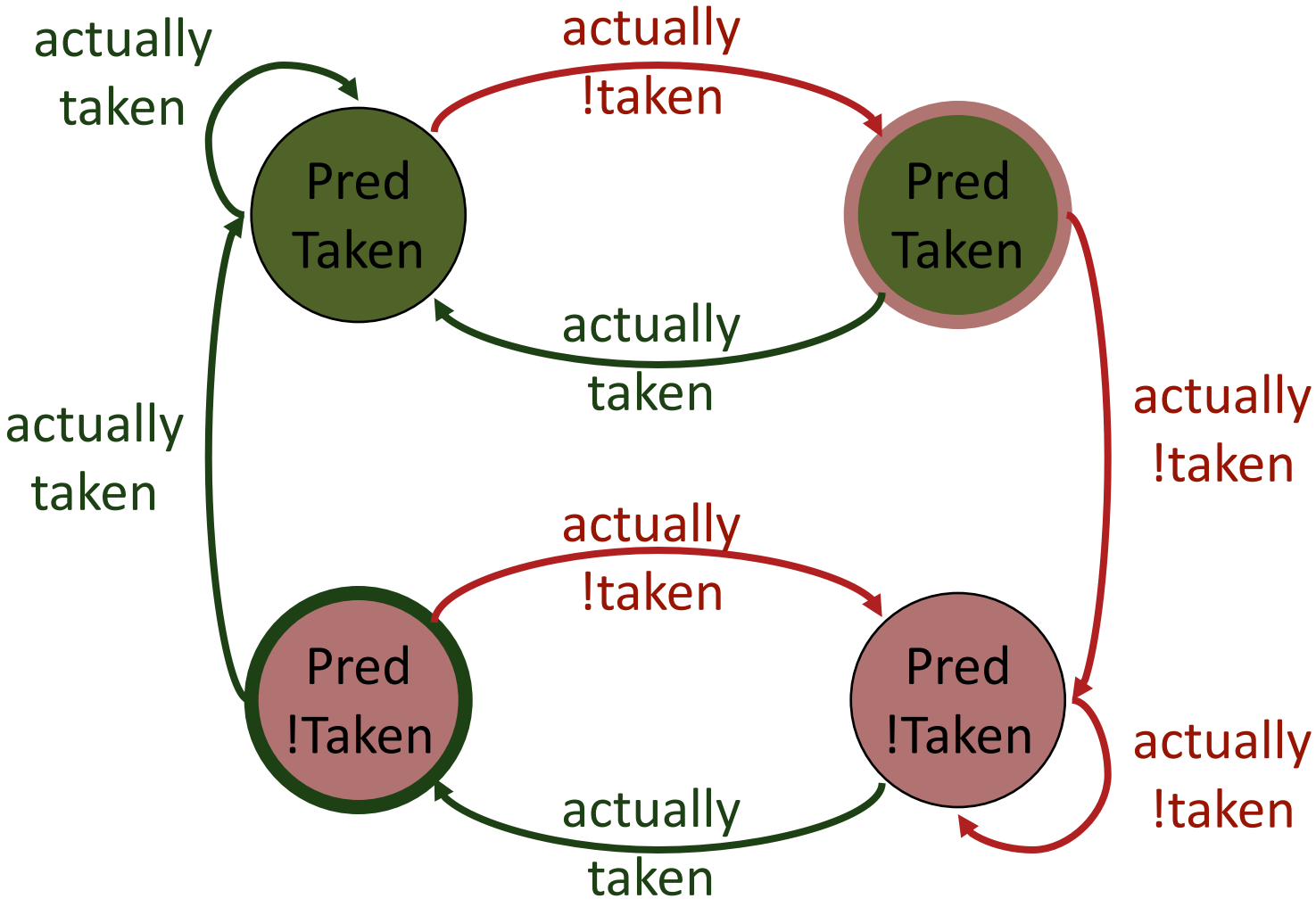
- Assume 2 BP bits per instruction
- Use saturating counter

On ¬taken ↓	On taken ↑			
		1	1	Strongly taken
		1	0	Weakly taken
		0	1	Weakly ¬taken
		0	0	Strongly ¬taken

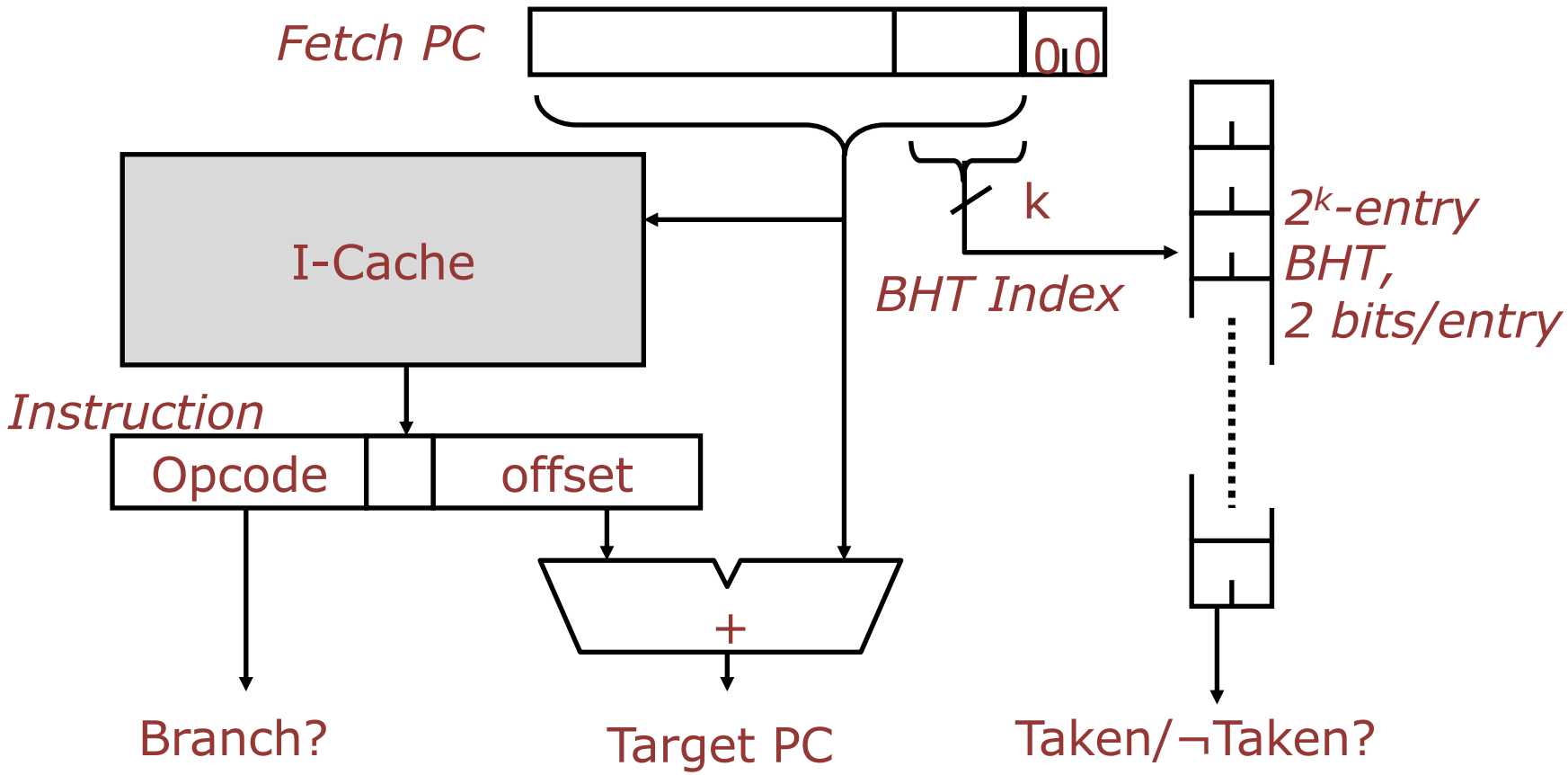
Two-bit Saturation Predictor



Forward Two-bit Predictor

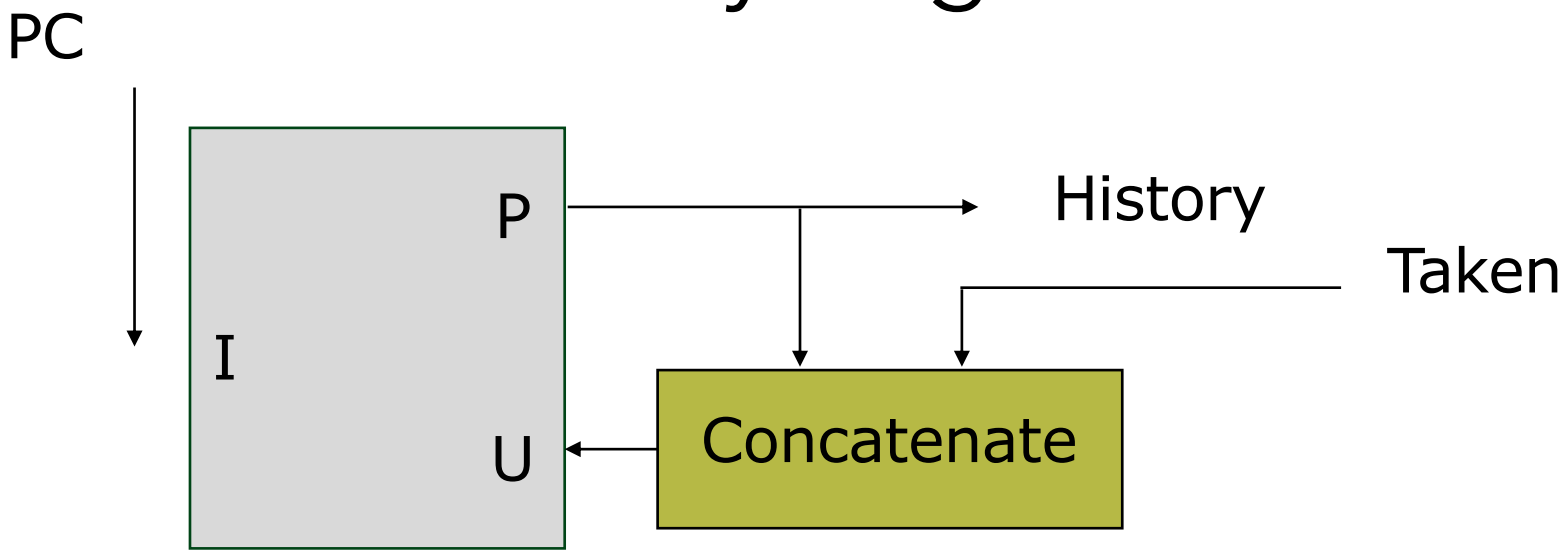


Branch History Table



4K-entry BHT, 2 bits/entry, ~80-90% correct predictions

History Register



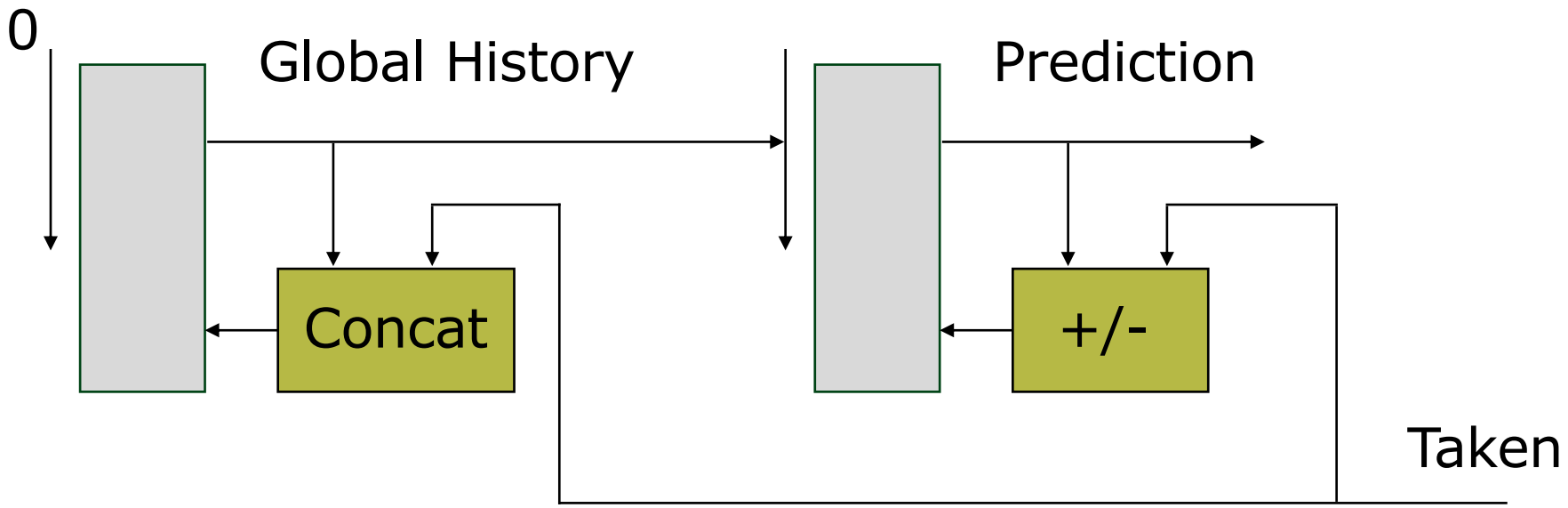
- History(PC, T) = P(PC; P || T)

Exploiting Spatial Correlation

```
if (x[i] < 7) then
  y += 1;
if (x[i] < 5) then
  c -= 4;
```

- If first condition false, second condition also false
- History register, H, records the direction of the last N branches executed by the processor

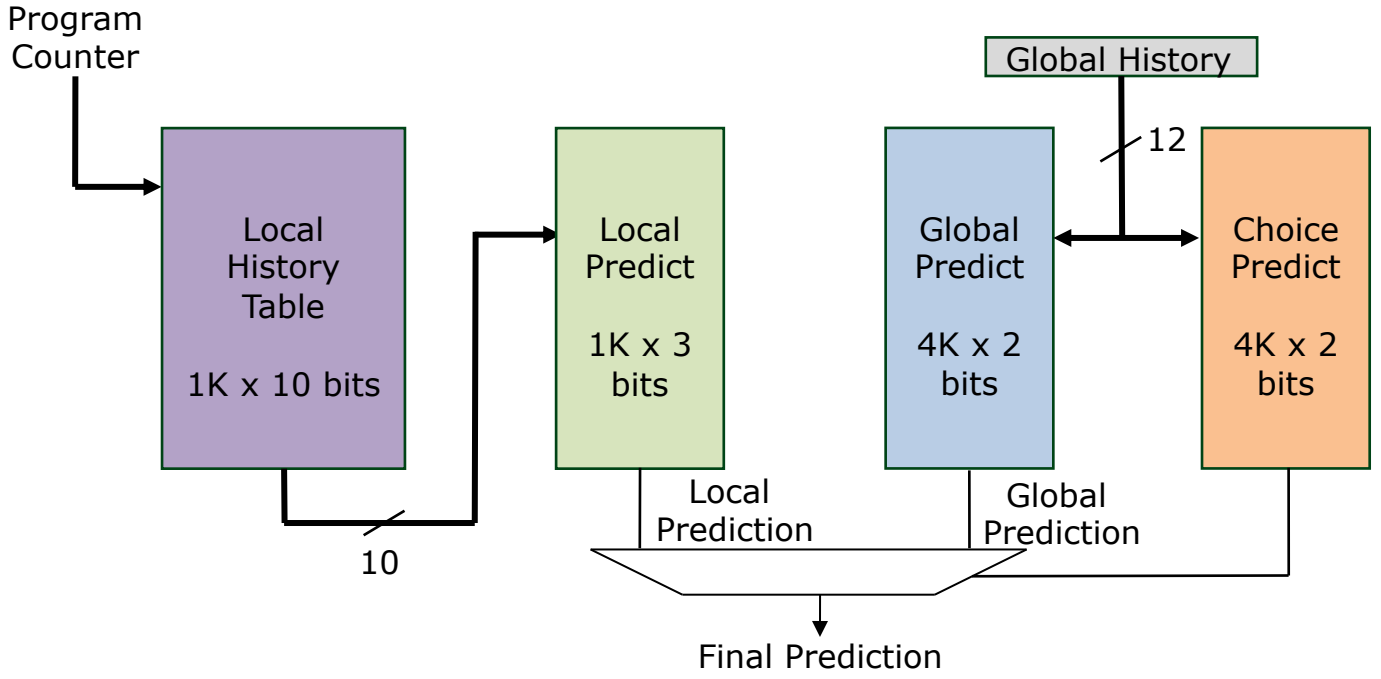
Two-level Predictor



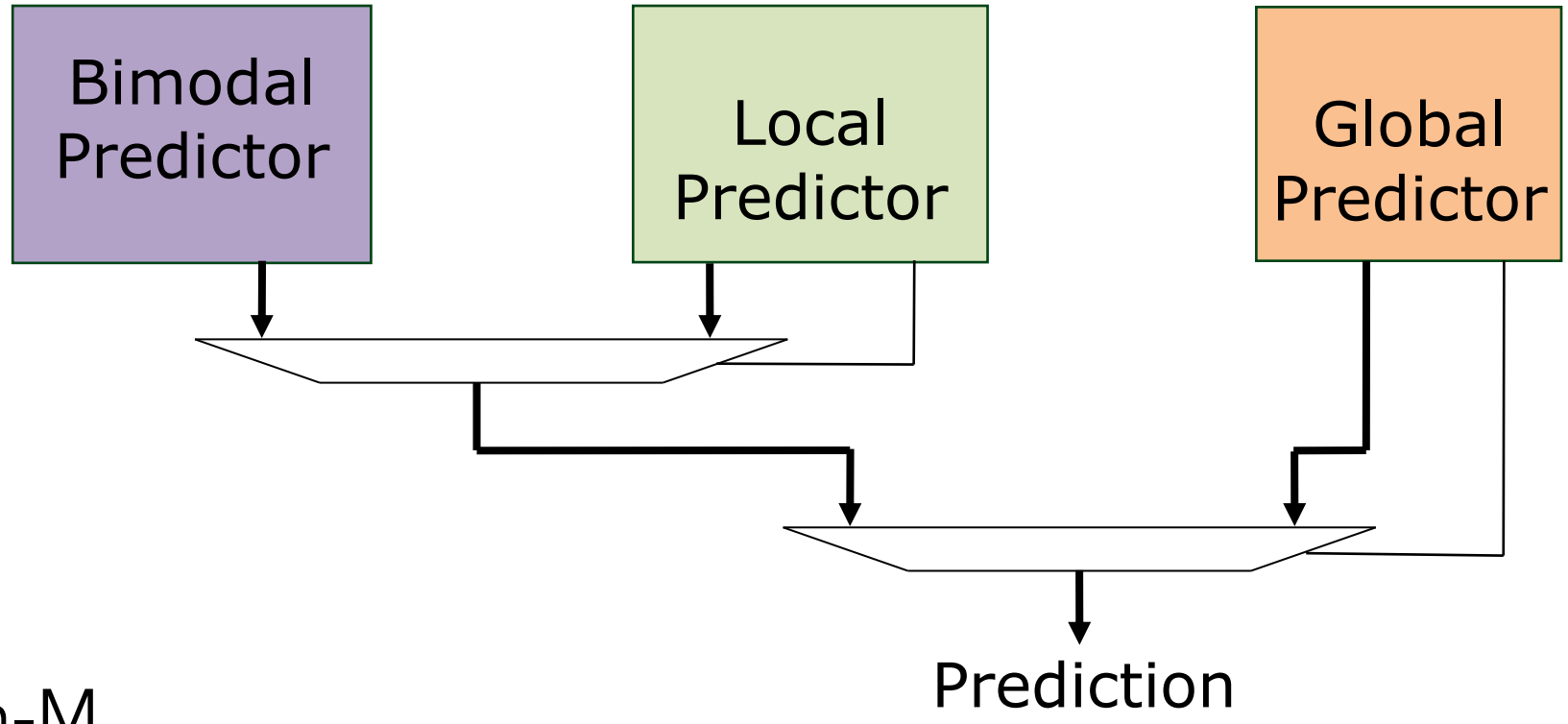
GHist(;T) = Counter(History(0, T); T)
Ind-Ghist(PC;T) = Counter(PC || Hist(GHist(;T);T))

Tournament Predictor

- Alpha 21264
  - Minimum branch penalty: 7 cycles
  - Typical branch penalty: 11+ cycles
  - 48K bits of target addresses stored in I-cache
  - Predictor tables are reset on a context switch

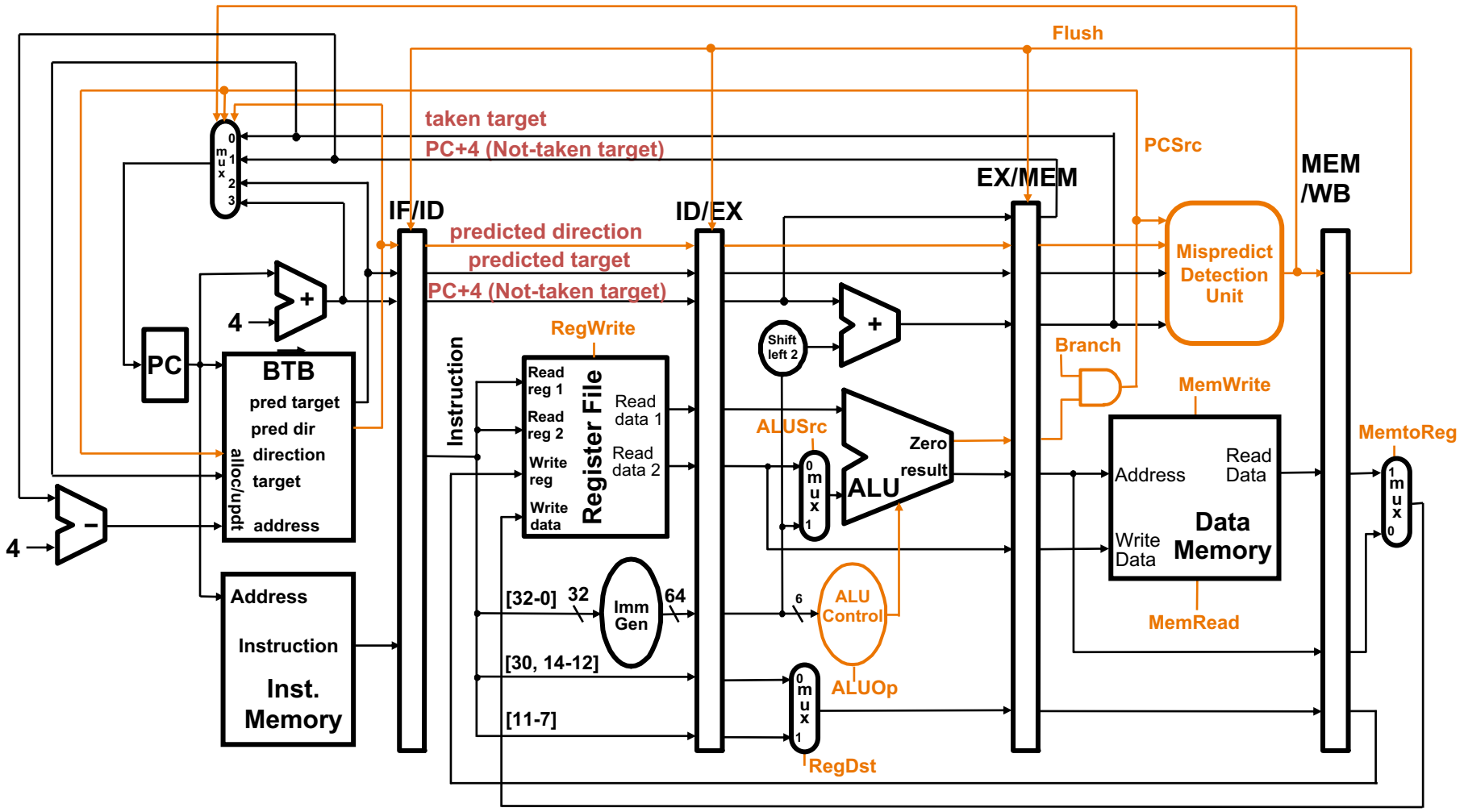


Hybrid Predictor

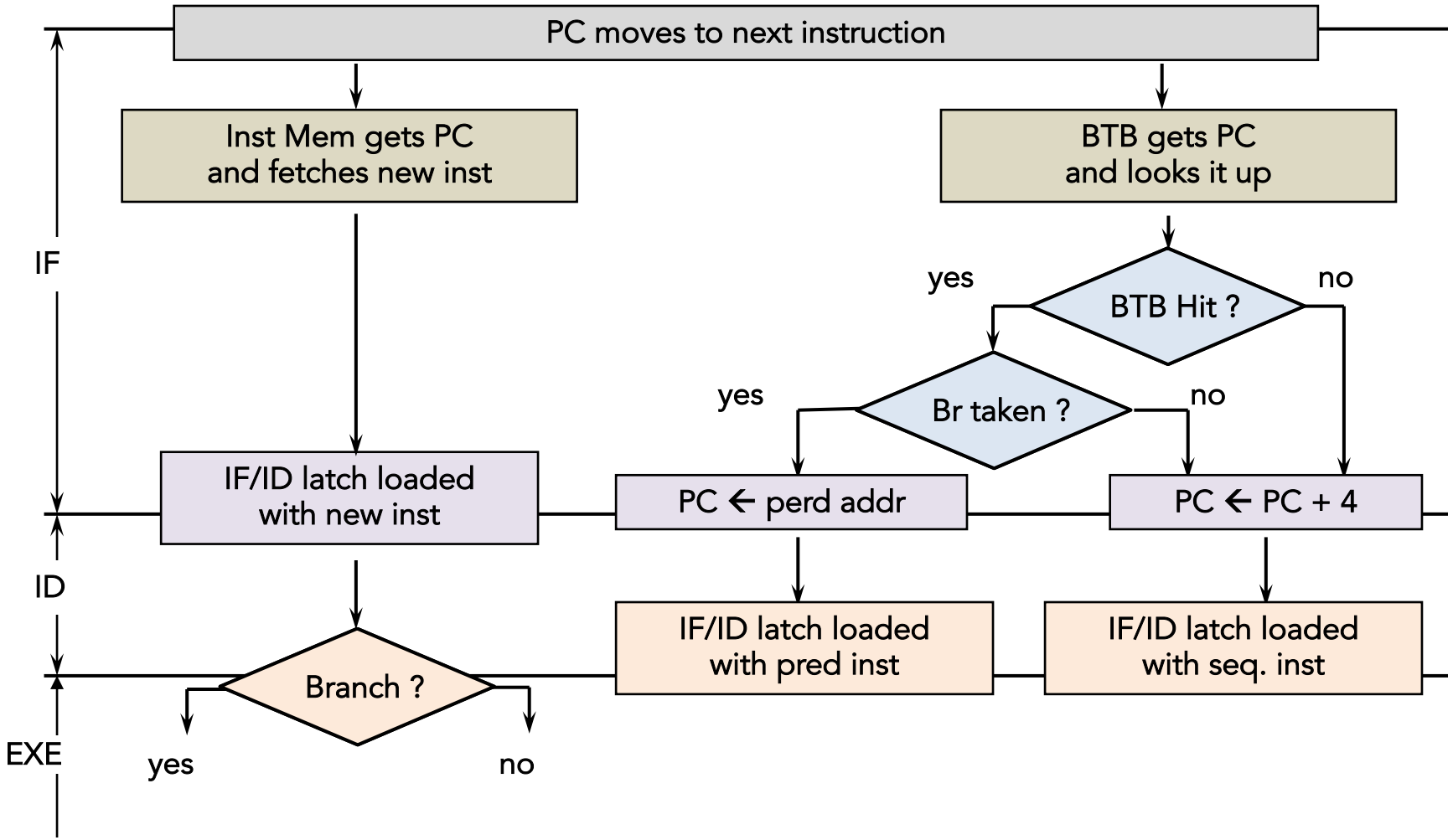


- Pentium-M
  - Hybrid, but uses tag-based selection mechanism

RISC-V Pipeline with a BTB



The pipeline Front-End with the BTB



Next learning Module

- Complex Pipelining: Superscalar Architecture