Advanced Computer Architecture

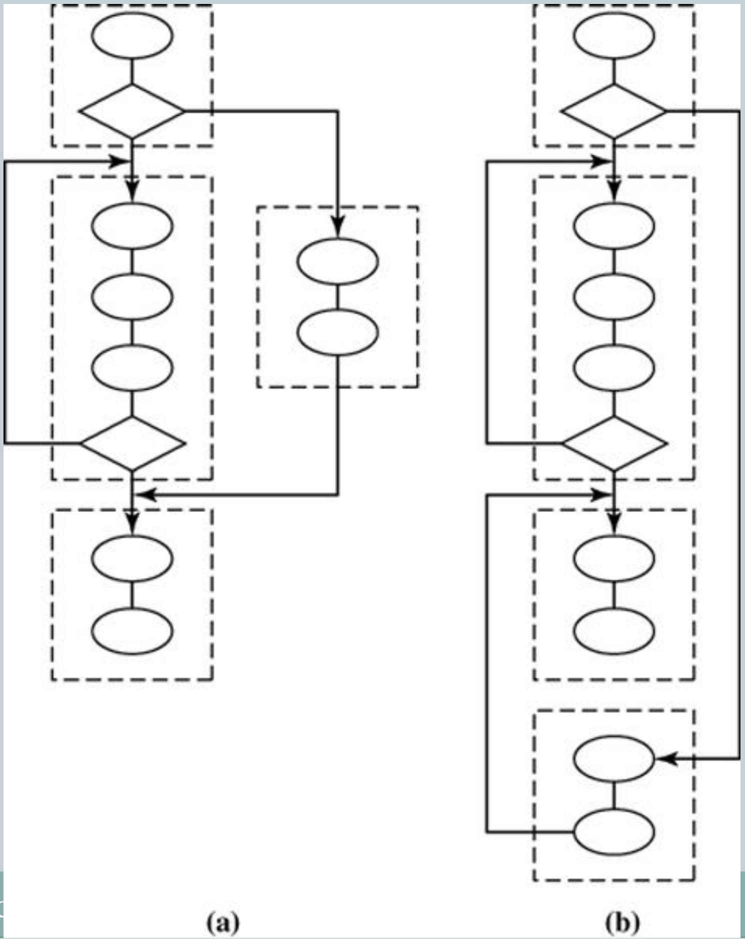
LECTURE 3: INSTRUCTION STREAMS [5.1]
BRANCH PREDICTION
EFFICIENT FETCHING
EFFICIENT DECODING
EFFICIENT DISPATCHING

JAN LEMEIRE

Program Control Flow

- Control flows irregularly through a program
  - conditional branches
  - unconditional branches
  - indirect branches (instruction specifing the address of the next instruction to execute)

- Problem: *how do we get instructions into the pipeline?*

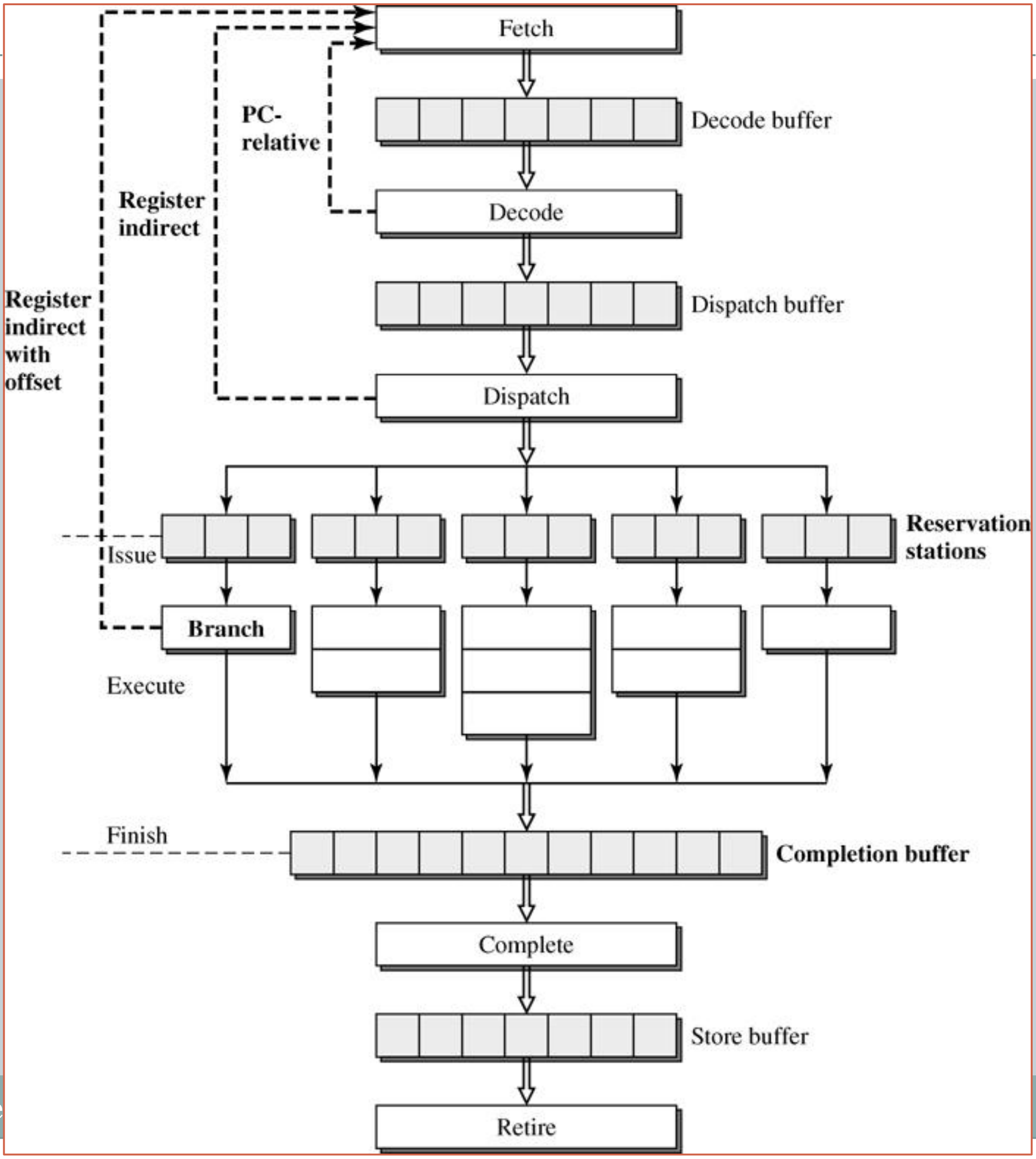


Instruction Streams

- Goal: issue as much as possible useful instructions as early as possible (to keep pipeline filled)
- Correct branch prediction is extremely important
  - Even more important when
    - pipelines become deeper (mispredication penalty)
    - width of architecture increases (superscalar)
    - branch instructions are more complex
- Efficient fetching & decoding is important
  - high bandwidth
  - high frequencies
  - also for CISC architectures!!!

Problems with Branches

- Potentially big pipeline bubbles



Branch Prediction

- Main idea: predict where control will be transferred, fetch and execute speculatively
- Observation
  - temporal locality in branching (loops)
  - can predict if we keep track of past
  - often can predict really well
    (+95% for some programs)
- Three tasks
  1. branch condition speculation/prediction
  2. branch target speculation/prediction
  3. branch mispredictions recovery

Static vs Dynamic Condition Prediction

- Static
  - one prediction per conditional branch in binary code
  - determined by software (or hardware convention)
  - statically at compile time[1]

- Dynamic
  - many predictions possible, based on local or global branching history
  - determined by hardware
  - dynamically at run time[1]

[1] In proper English, one writes "compile-time optimization" and "optimization at run time".

Static Branch Condition Prediction (1)

- Determine statically for each branch what its predicted condition will be (taken or not-taken)

- Condition determined by
  - conventions
  - hint bit in the instruction encoding
- Three options
  - rule-based
  - program-based
  - profile-based

Static Branch Condition Prediction (2)

- Pro
  - Cheap, not complex, little hardware
- Con
  - Not adaptive to program input
  - Not adaptive to dynamic program behavior
- Interesting for
  - hybrid static-dynamic predictors
  - low-power embedded processors
  - compiler optimization such as code layout

Static Branch Condition Prediction (3)

- Rule-based
  - Never taken
    - simple hardware, sequential fetching
  - Always taken
    - more complex hardware, need to know (PC-relative) target address
    - need to fill branch delay slot (hard in OoO processors, hard for compilers)
  - Backward taken, forward not taken (BTFNT)
    - only backward branches (corresponding to loops) are mostly taken
    - for others, compiler can play with code layout

  - In common: based on low-level (machine code) properties

Static Branch Condition Prediction (4)

- Program-based
  - Requires a hint bit in instruction encoding
  - Features and structure of the source language determine hints
    - loop branch: predict taken
    - NULL-test for pointers: predict non-NULL
    - pointer comparison: predict not equal
  - More accurate than rule-based because of high-level decision logic

Static Branch Condition Prediction (5)

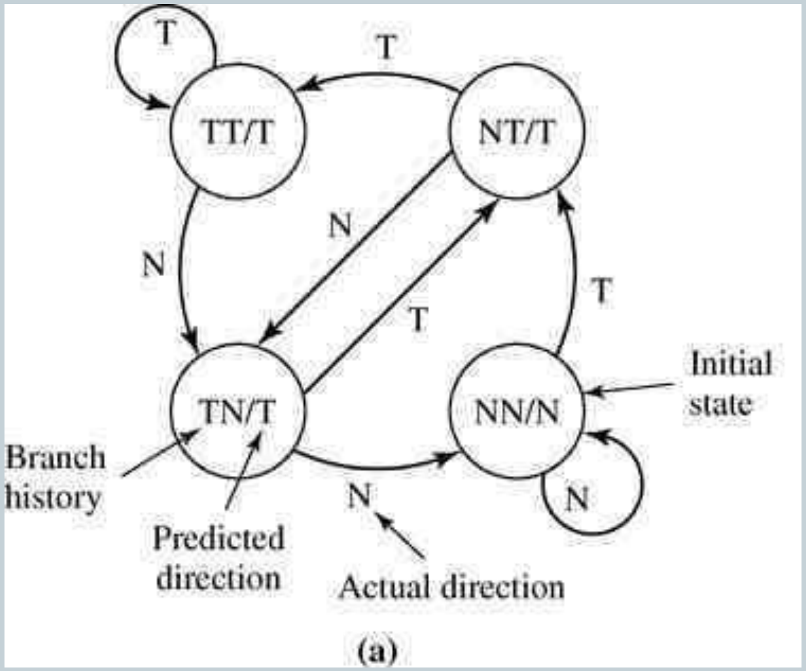
- Profile-based
  - Requires a hint bit in instruction encoding
  - Profile application to collect condition statistics
  - Feed back the statistics to 2nd compiler run, fills in bits
    - hint is taken when branch was taken more than 50% of the time
  - More accurate than program-based because program-based rules can be tuned
  - Requires representative inputs

Dynamic Branch Condition Prediction (1)

- More accurate 80%-97% (↔ static 70%-80%)
- Some branches are hard to predict statically, but easy dynamically
  - First half of program not-taken, second hald taken
  - Alternating taken and non-taken
  - Input-dependent branches
- Adapts to dynamic behavior of a program
  - Prediction depends on context of branches
- Common in all predictors
  - Finite state machines keep track of recent histories to determine current prediction: pattern history tables
  - Some indirection scheme to choose a particular finite state machine

Dynamic Branch Condition Prediction (2) [225]

- Finite state machines : 2-bit saturating counters
  - Design decision: *favoring taken or not-taken*
  - 2-bit states
  - lookup
    - 00: predict not-taken
    - 01: predict not-taken
    - 10: predict taken
    - 11: predict taken
  - update
    - taken: +1 (saturating arithmetic)
    - not taken: -1



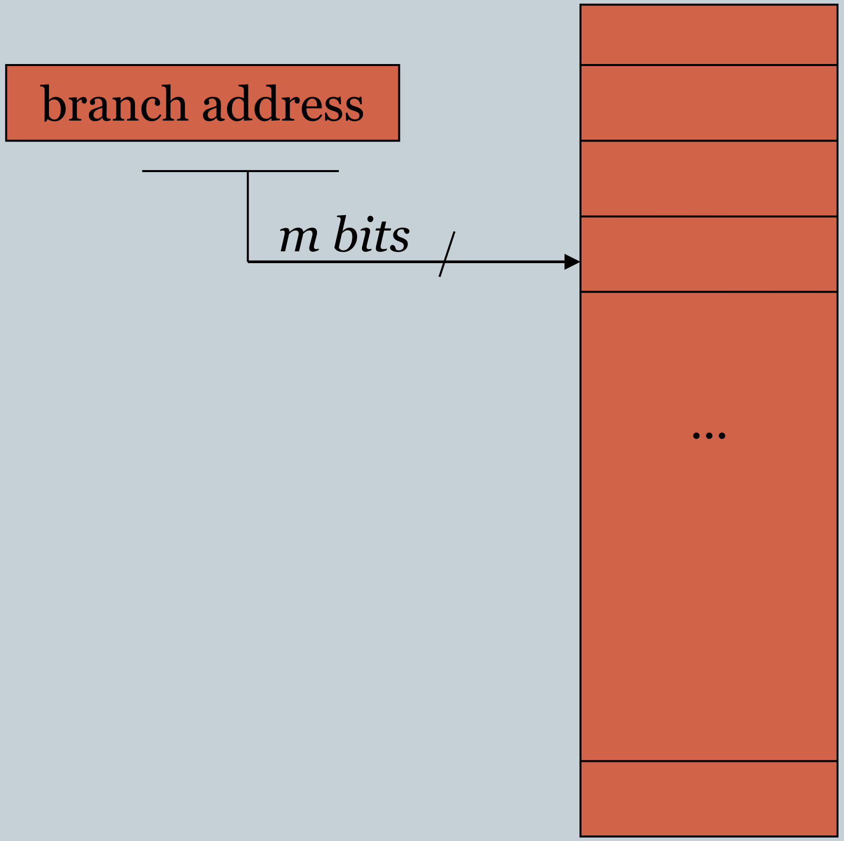
Dynamic Branch Condition Prediction (3)

- 2-bit saturating counters: suppose the following sequence of branch directions

branch direction	*state before*	*updated state*	*prediction*
0	00	00	**0**
0	00	00	**0**
1	00	01	**0** ⚡
1	01	10	**0** ⚡
1	10	11	**1**
1	11	11	**1**
0	11	10	**1** ⚡
1	10	11	**1**
1	11	11	**1**
0	11	10	**1** ⚡
0	10	01	**1** ⚡

Dynamic Branch Condition Prediction (4)

- Basic bimodal branch predictor



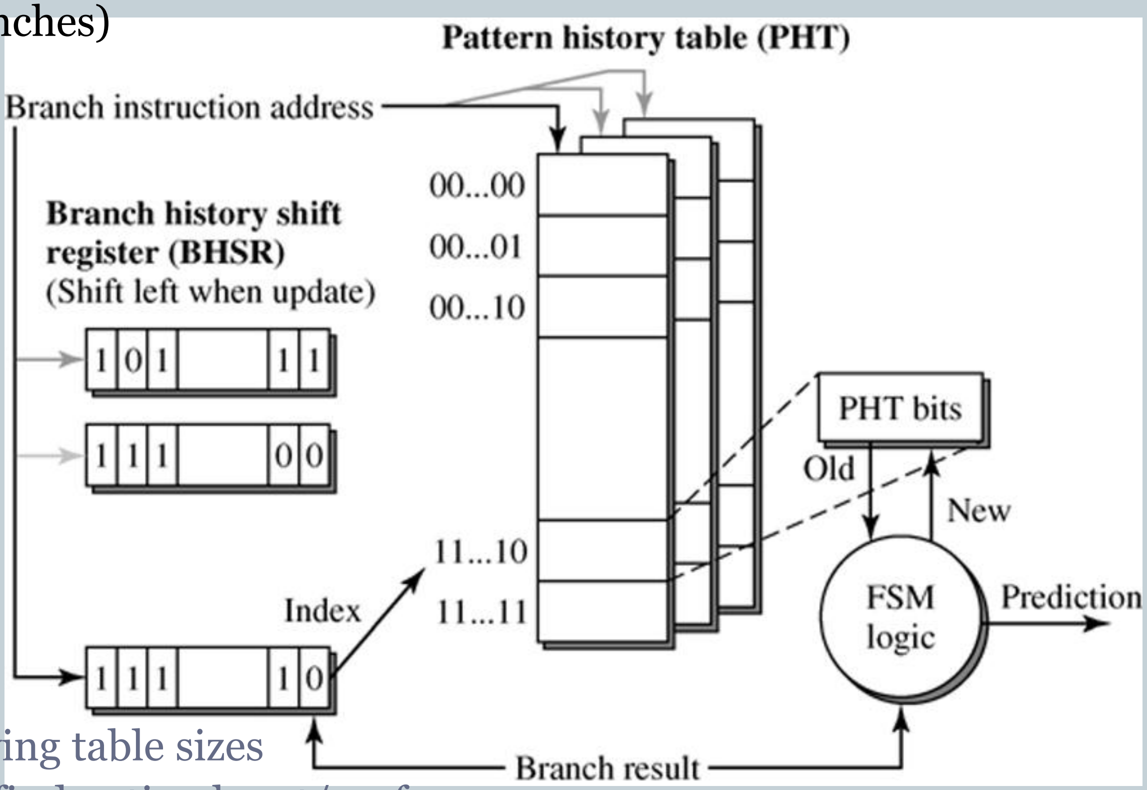
2^m 2-bit saturating counters

as in a cache, multiple (branch) addresses are mapped
- onto the same line
- onto the same FSM

this can result in aliasing

Dynamic Branch Condition Prediction (5)

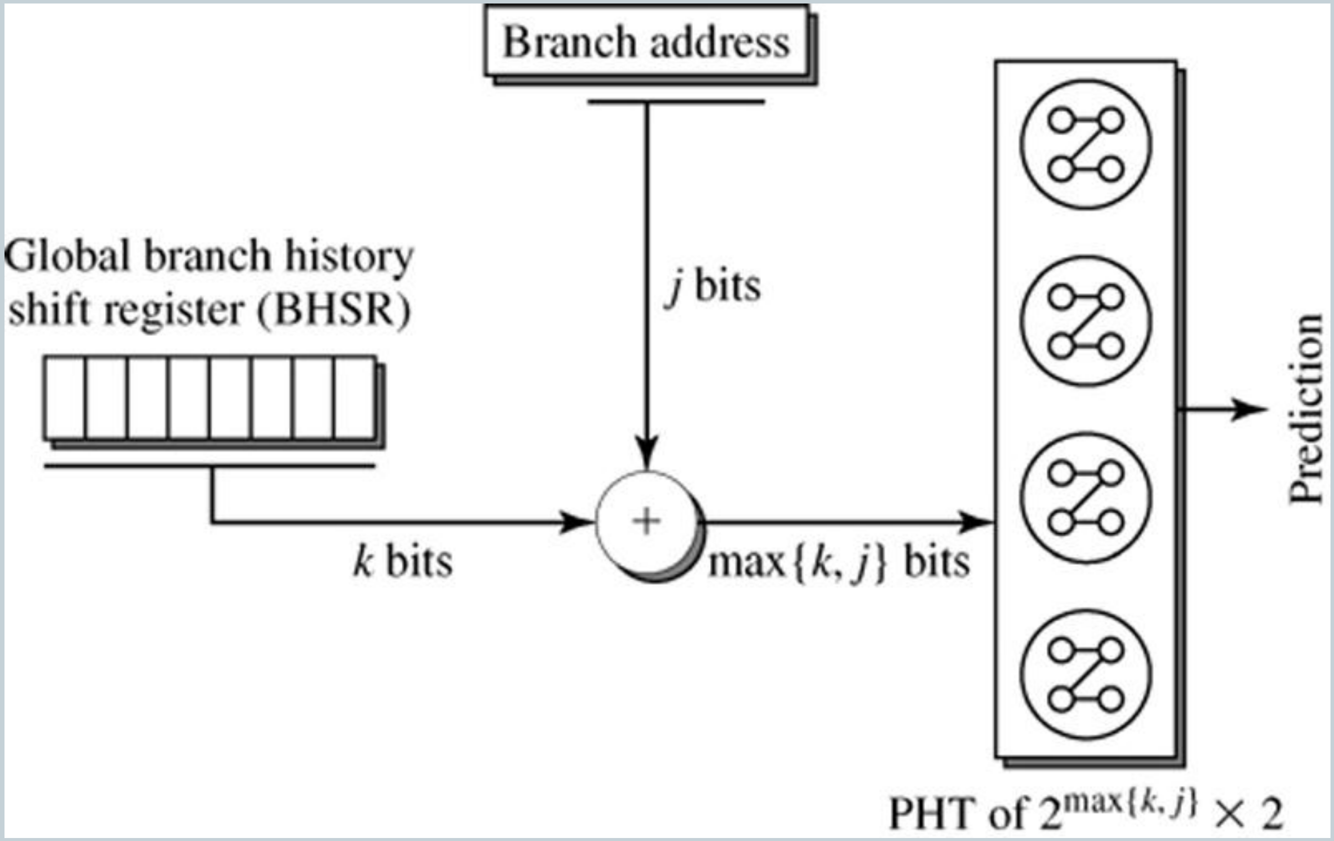
- Two-level adaptive predictors
- BHSR (recently executed branches)
  - global (G)
  - individual (P)
- PHT
  - global: 1 table (g)
  - individual: 1 table per branch (p)
  - shared: 1 table for a small number of branches (s)
- history-based FSM
  - adaptive (A)
- Allows several designs
  - GAg, PAg, PAs, ... with varying table sizes
  - a large design space, try to find optimal cost/performance


Pattern history table (PHT)
Branch instruction address
Branch history shift register (BHSR) (Shift left when update)
1 0 1 1 1
1 1 1 0 0
1 1 1 1 0
Index
00...00
00...01
00...10
11...10
11...11
PHT bits
Old
New
FSM logic
Prediction
Branch result

When branches correlate with behavior of other branches

Dynamic Branch Condition Prediction (6) [234]

- Alternative: gshare (McFarling, 1993)


Branch address
Global branch history shift register (BHSR)
j bits
k bits
+
max{k, j} bits
Prediction
PHT of $2^{\max\{k,j\}} \times 2$

Dynamic Branch Condition Prediction (7)

- Two-level adaptive
- Gshare
- work because of correlation between branches
  - e.g., branches that test same variable
  - or just statistically correlated
  - best with global predictors
- works because of recurring patterns
  - best with local (individual) predictors
- different parameters for different correlations and patterns

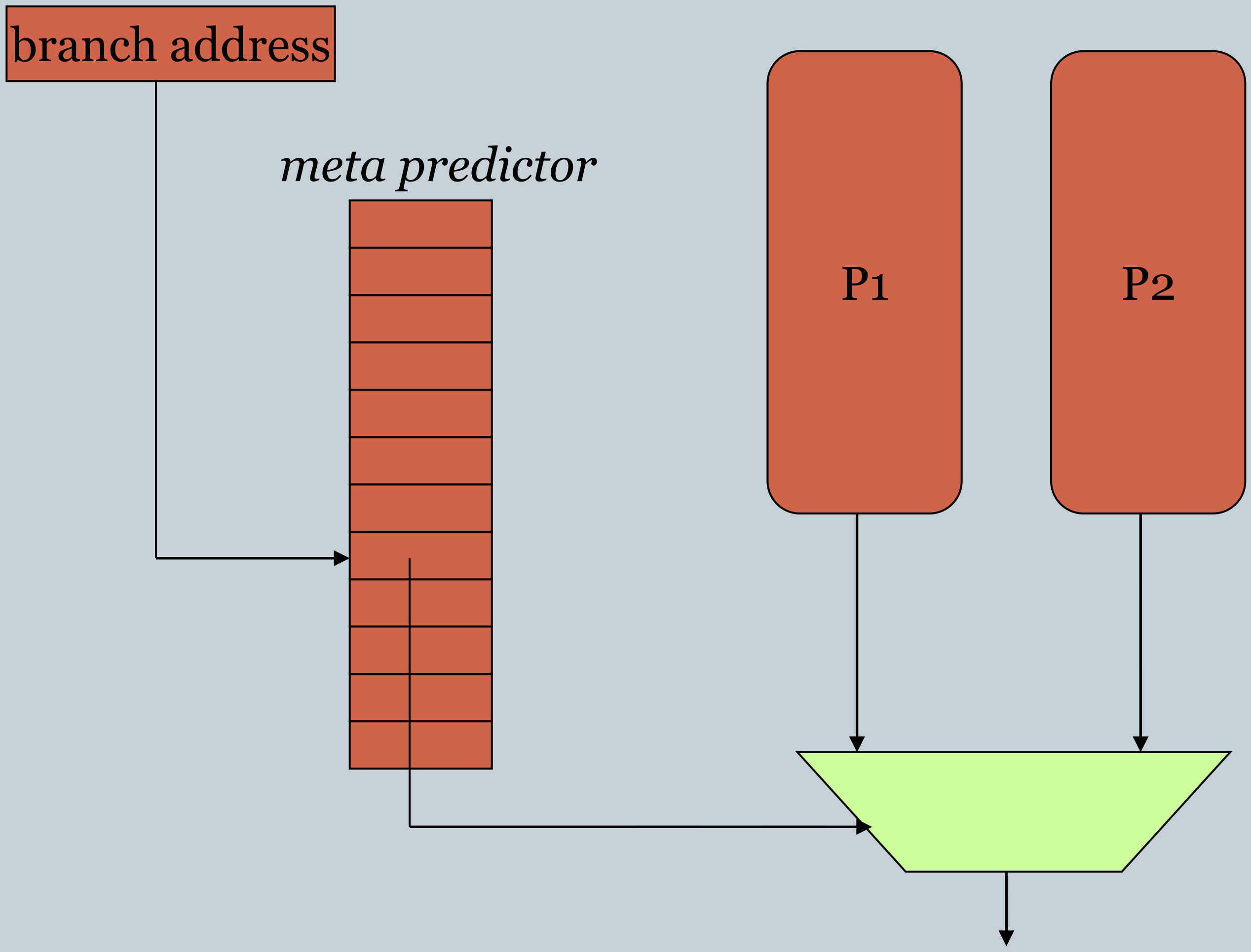
Hybrid Branch Condition Prediction (1)

- Sometimes prediction still goes wrong
  - Branch can be hard to predict
    - Predictor is still being trained
    - Some branches behave truly random
  - Tables are limited in size
    - Interference, conflicts or *aliasing*
      - Can be negative, neutral or positive (correlated jumps)
    - Two possible reasons
      - Tables too small for number of branches
      - Hash function maps branches to same lines
  - Behavior or branch does not fit type of predictor
- Solution: hybrid branch predictor

Hybrid Branch Condition Prediction (2)

- General idea: combine different predictors
- Some branches will be predicted better by one of the predictors
- Remember which predictor is best for each branch
- Several types
  - tournament
  - static
  - branch classification
  - multi-hybrid
  - etc.

Tournament Predictor (1) [491]

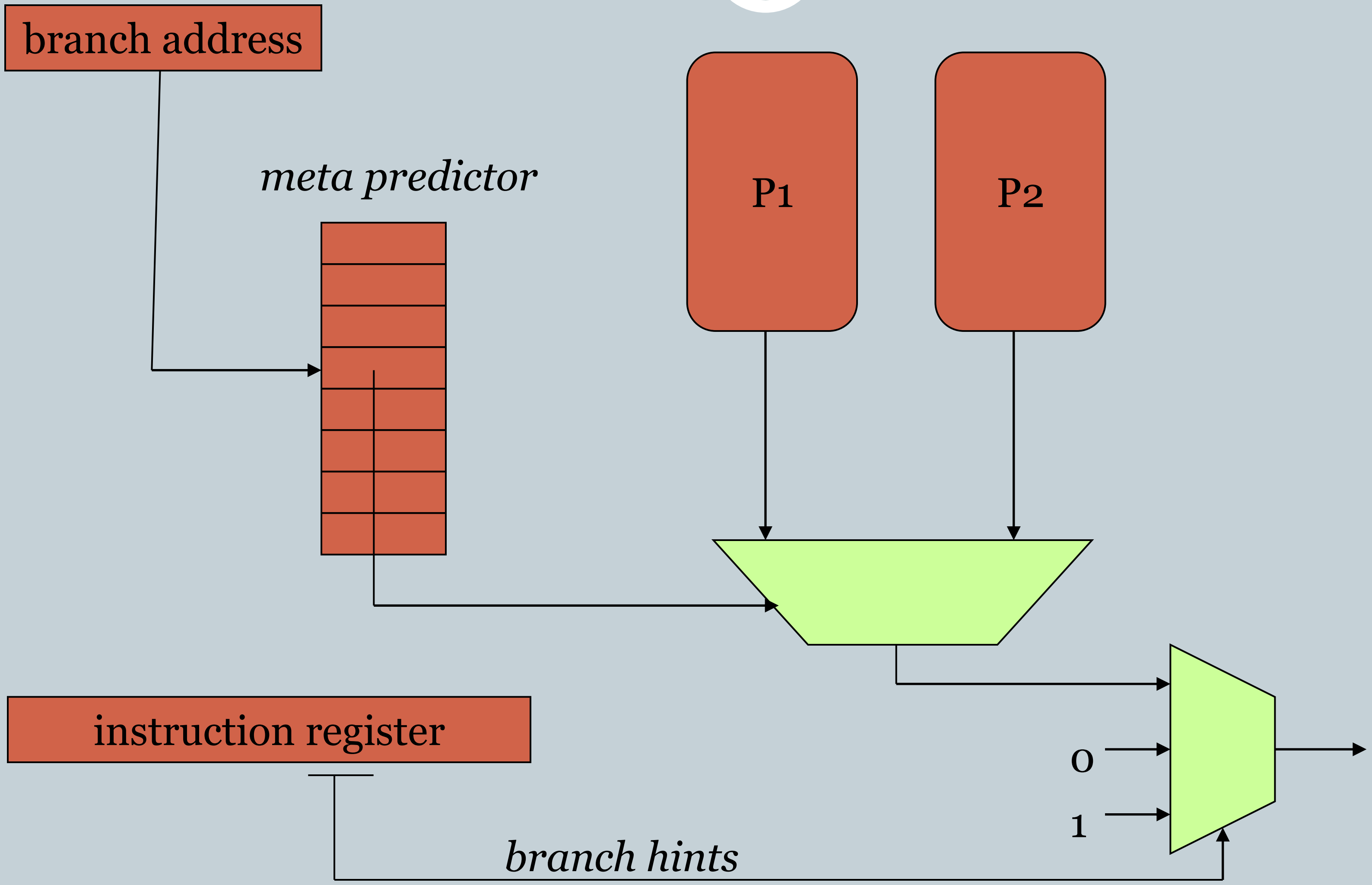


Tournament Predictor (2)

- Meta predictor is two-bit counter that decides which predictor is used
  - If $<2 \rightarrow$ P1; if $\geq 2 \rightarrow$ P2
- Update meta predictor
  - Do nothing if both predictions correct
  - Decrement if P1 correct and P2 incorrect
  - Increment if P1 incorrect and P2 correct
- Update both predictors on update

- Typically have a global and a local predictor

Branch Classification (1)



Branch Classification (2)

- Use static predictor for branches that predict well statically (e.g., +95% changes)
- Predict other branches dynamically
- Pro:
  - less branches in tables
  - hence less aliasing and better performance
- Con:
  - hints are available in ID stage only

Example 1: Alpha 21264

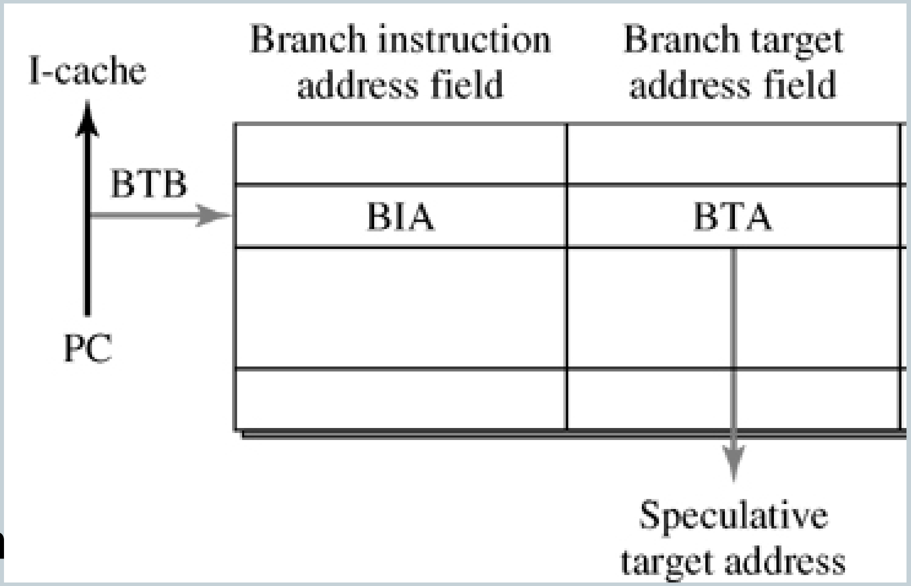
- Hybrid predictor consisting of
  - Local PAg
    - 1st level: 1K 10-bit elements
    - 2nd level: 1K 3-bit saturating counters
  - Globale GAg
    - 4K 2-bit saturating counters
    - 12 bits global branch history
  - Meta predictor
    - 4K 2-bit saturating counters
    - Indexed as global predictor

Vb. 2: IBM POWER4

- Hybride predictor consisting of
  - Bimodal predictor
    - 16K 1-bit saturating counters
  - Gshare predictor
    - 16K 1-bit saturating counters
    - 11 bits global history
  - Meta predictor
    - 16K 1-bit saturating counters
    - Indexed like gshare predictor

Branch Target Buffers [226]

- need to know in time from where to fetch
  - when condition is speculated for conditional branches
  - always for non-conditional branches
- branch target buffer (or branch target access cache)
  - cache
  - indexed by branch instruction address
  - lookup returns branch target address
- if address is not present, assume not taken
- very early in pipeline!
- update on retirement
- store all branches or only taken

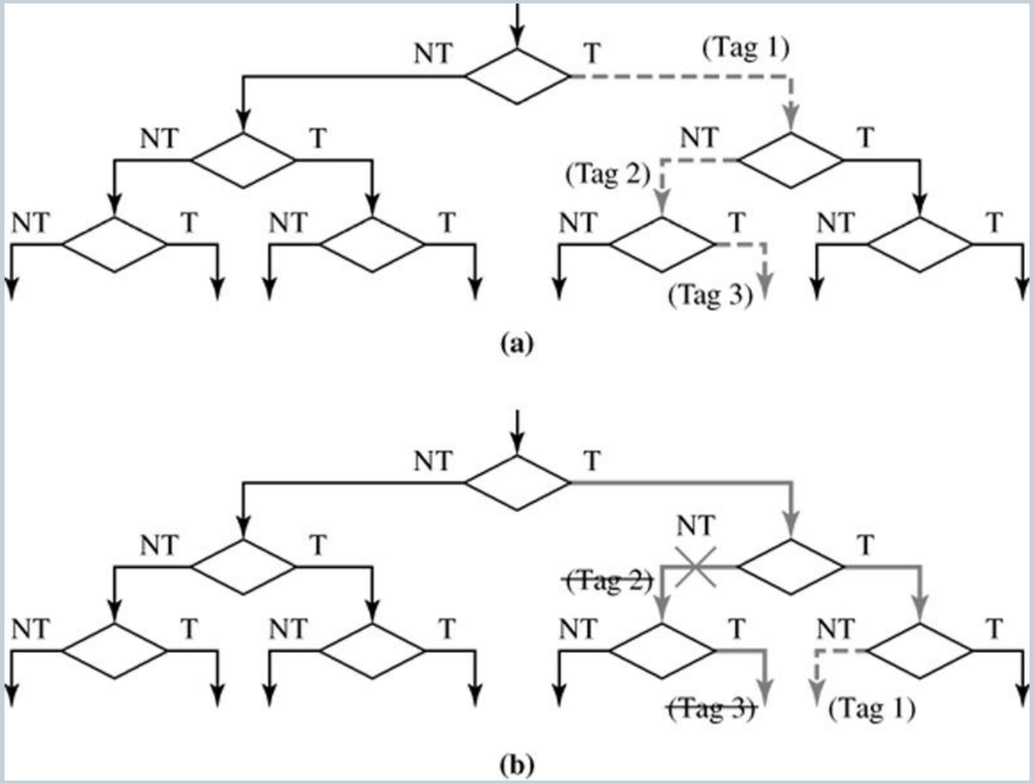

I-cache
BTB
PC
Branch instruction address field
Branch target address field
BIA
BTA
Speculative target address

Other Branch Prediction Techniques

- many extensions exist
- trace cache (in a couple of slides)
- return address stack
  - keep a small stack of return addresses
  - push on call
  - pop return address on return
- to save time, different tables and caches are accessed together *and concurrently*, then choice is made !

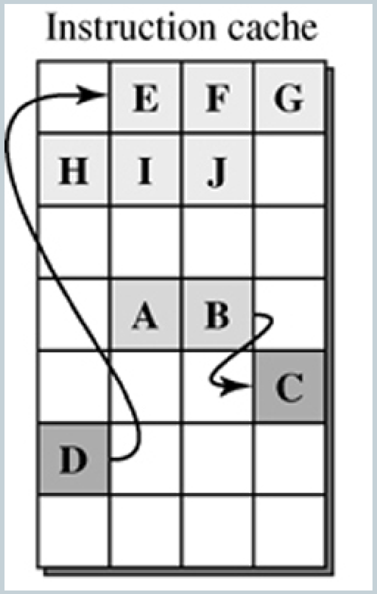
Misprediction Recovery [218-219]

- Speculative instructions are tagged (with tag specific for branch)
- When branch is really executed, prediction is validated
- Upon misprediction
  - mispredicted instructions are discarded
  - fetching at correct place is initiated



Efficient Instruction Fetching (1) [504]

- Branch predictors reduce control flow dependencies
- Still fetching instructions from I$ in program order
- Problem 1: what if fetch block spans more than one I$ line
- Problem 2: together with a taken branch, non-executed instructions may be stored in cache

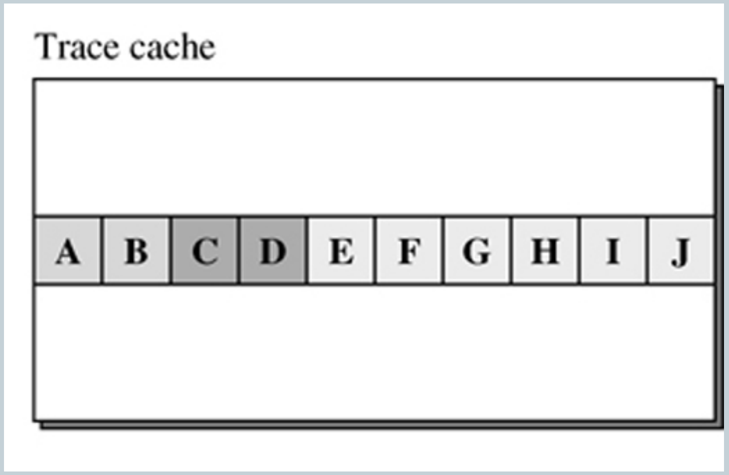


Efficient Instruction Fetching (2)

- Solution 1: Compiler optimizes code layout to place basic blocks at good cache alignment
  - problem: code generation becomes microarchitecture-dependent
  - far from optimal
- Solution 2: Auto-realignment hardware

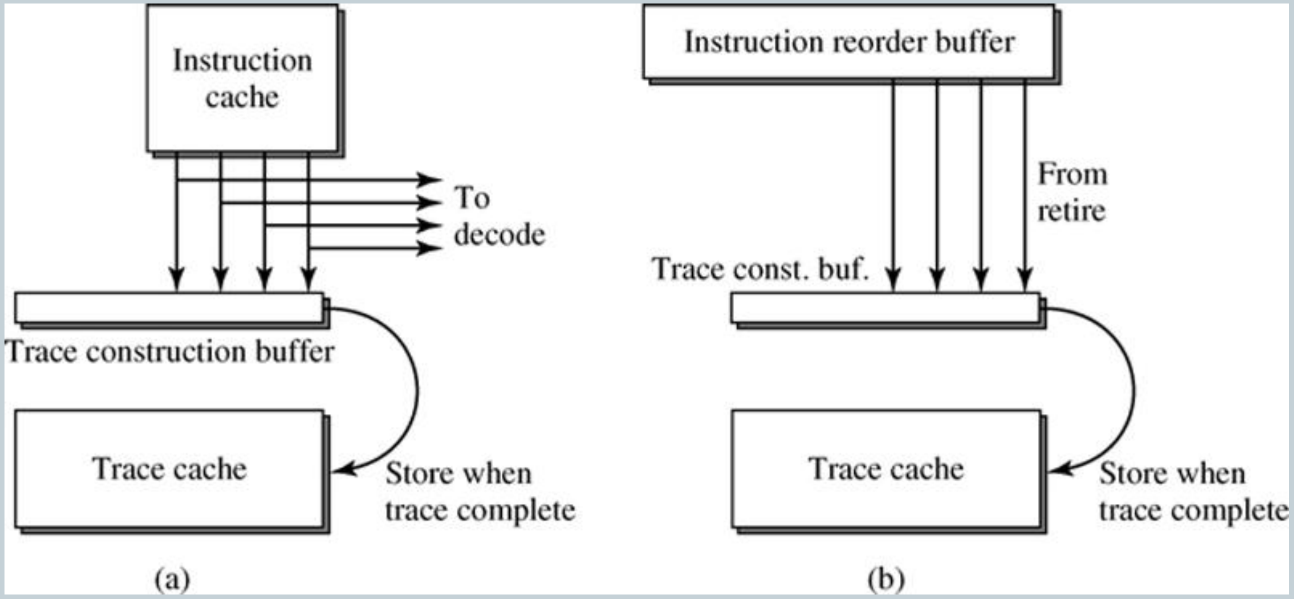
Efficient Instruction Fetching (3) [506]

- Alternative: trace cache
  - instead of storing static instructions based on their address
  - store dynamic instructions (traces) based on their address and on branch outcomes, higher bandwidth can be obtained



Efficient Instruction Fetching (4)

- Alternative: trace cache
  - instead of storing static instructions based on their address
  - store dynamic instructions (traces) based on their address and on branch outcomes, higher bandwidth can be obtained
  - fetch-time storing or completion-time storing


Instruction cache
To decode
Trace construction buffer
Trace cache
Store when trace complete
(a)
Instruction reorder buffer
From retire
Trace const. buf.
Trace cache
Store when trace complete
(b)

Efficient Instruction Fetching (5) [509]

- High frequency fetching
  - many techniques to speculate where to fetch
  - large tables of precise predictors are slow (multiple cycles)
  - overriding branch predictors (©2000)
    - very accurate predictors are complex and slow
    - hence first use a simple, single-cycle predictor
    - override it one or more cycles later by complex, multi-cycle predictor

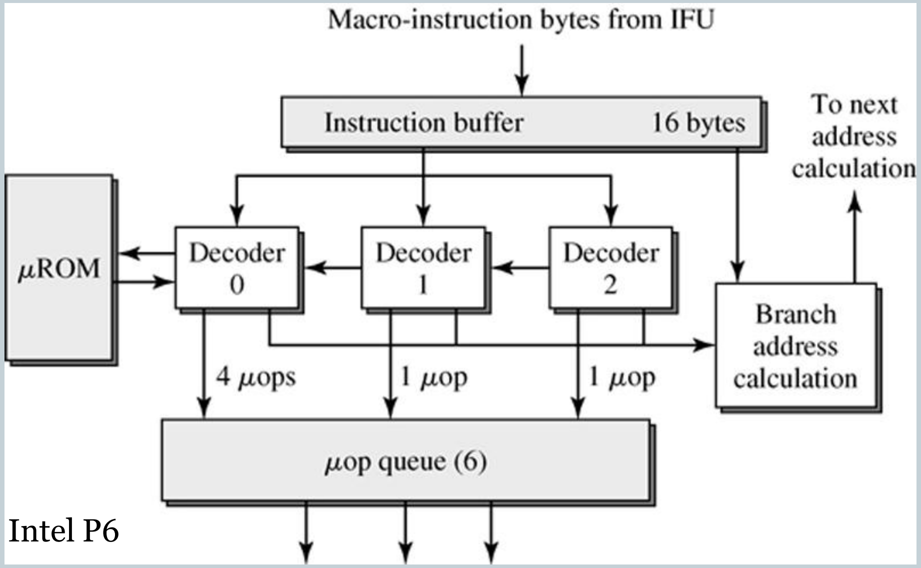
Efficient Instruction Decoding (1) [195]

- Decoding determines
  - what the individual instruction types in the fetch group are
  - what their types are, operands, etc…
  - identify dependencies & branch instructions

  => Comparators & multi-ported registers
- Complexity depends on
  - ISA
  - width of superscalar pipeline
  - frequency to be obtained
- RISC: easy
  - fixed instruction width
  - limited nr of instruction types
- CISC: much more complex -> several stages

Efficient Instruction Decoding (2)

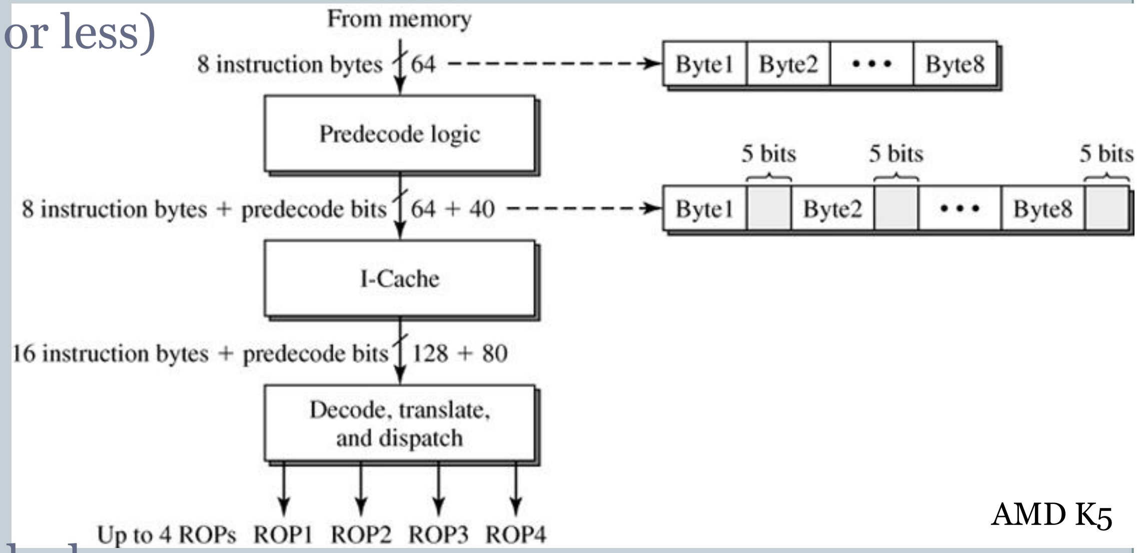
- CISC instruction widths vary
- Hence decoding is very difficult
- Requires multiple pipeline stages
  - Early on in pipeline -> bad for branch misprediction penalties
- Is very hard to parallelize (sequential dependence on width)
- Need to generate micro-ops
  - Intel: micro-operations
  - AMD: RISC operations
  - Intel: 1.5 – 2 micro-ops/instruction



Intel P6

Efficient Instruction Decoding (3)

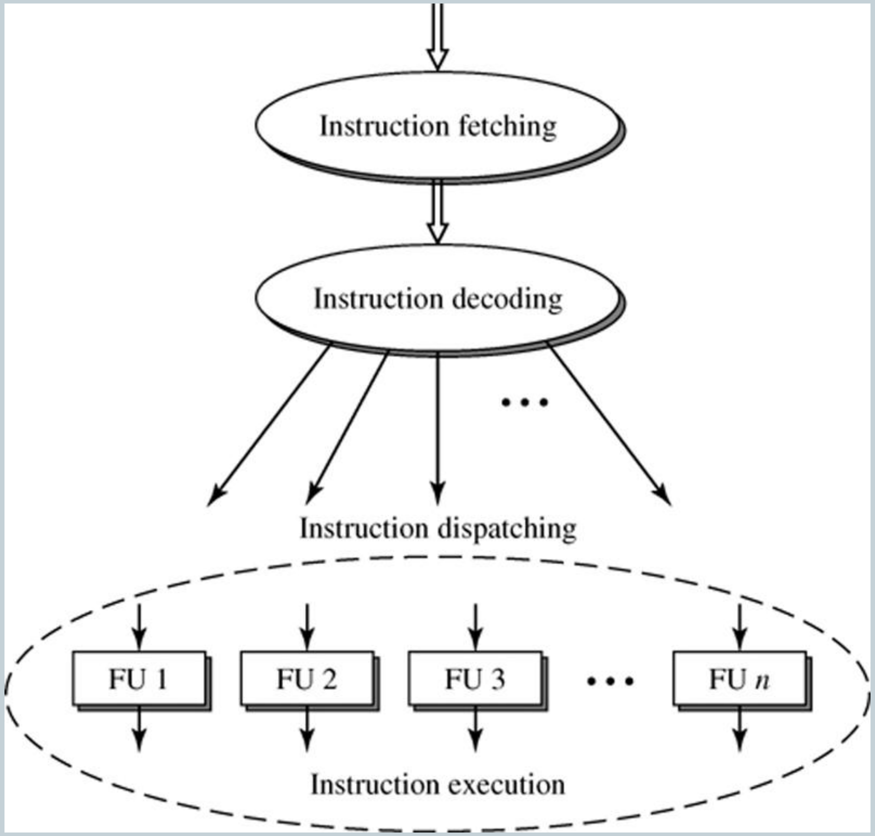
- Alternative: predecoding [198]
  - (partly) decode when instruction are brought in from memory
  - Intel: trace cache in Pentium 4
  - AMD: regular I$
- Pro:
  - decoding only once (more or less)
  - much easier decoding
- Con:
  - larger caches
  - higher cache-memory latency
- RISC? Yes, also
  - to identify branches
  - independent ops in fetch block



AMD K5

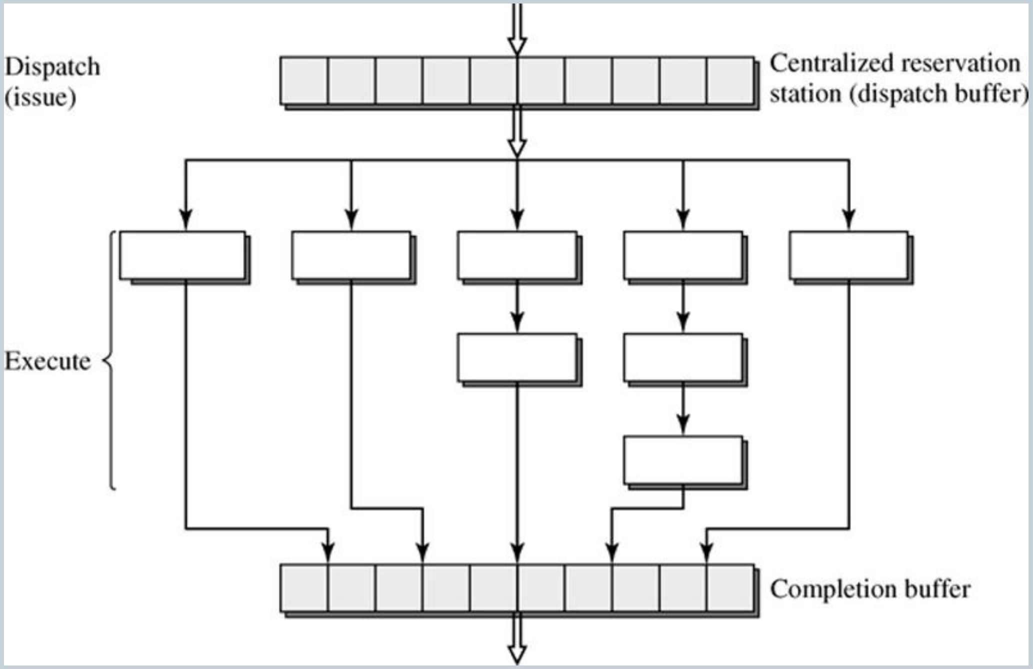
Efficient Instruction Dispatching (1)

- Routing instructions to functional units
- Decentralizes
  - previous pipeline stages are centralized
  - FU pipelines are decentralized
- Parallel
  - types are already known
- Dispatch instructions to
  - reservation station(s)
  - temporary buffers
  - waiting for operands



Efficient Instruction Dispatching (2)

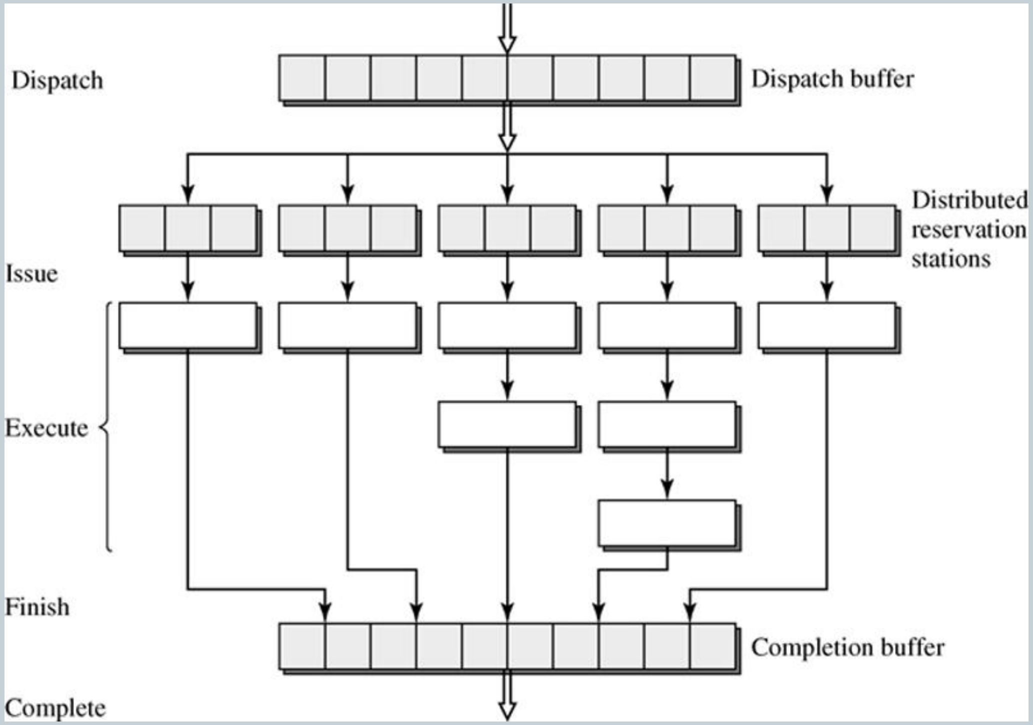
- Centralized reservation station
- Pro
  - less blocking
  - higher IPC
- Con
  - long and complex wiring
  - complex decision logic
- Example
  - Intel Pentium Pro



Efficient Instruction Dispatching (3)

- Distributed reservation station
- Pro
  - smaller structure, less wiring
  - simple decision logic
  - low hardware complexity
- Con
  - Worse overall utilization
  - Saturation/blocking possible
  - lower IPC
- Example
  - IBM PowerPC 650

- Also combinations possible



Efficient Instruction Dispatching (4)

- Terminology

- Dispatch
  - push instruction into reservation station
  - in OoO architecture: push into reorder buffer
  - decentralized reservation stations: routing to correct station

- Issue
  - select an instruction from reservation station
  - start its execution in the functional unit (pipeline)

Acknowledgement

- Thanks for (parts of) slides
  - Bjorn De Sutter
  - Lieven Eeckhout
  - Mikko H. Lipasti
  - James C. Hoe
  - John P. Shen