Superscalar Architecture

Organization | Fetch | Decode | Scheduling

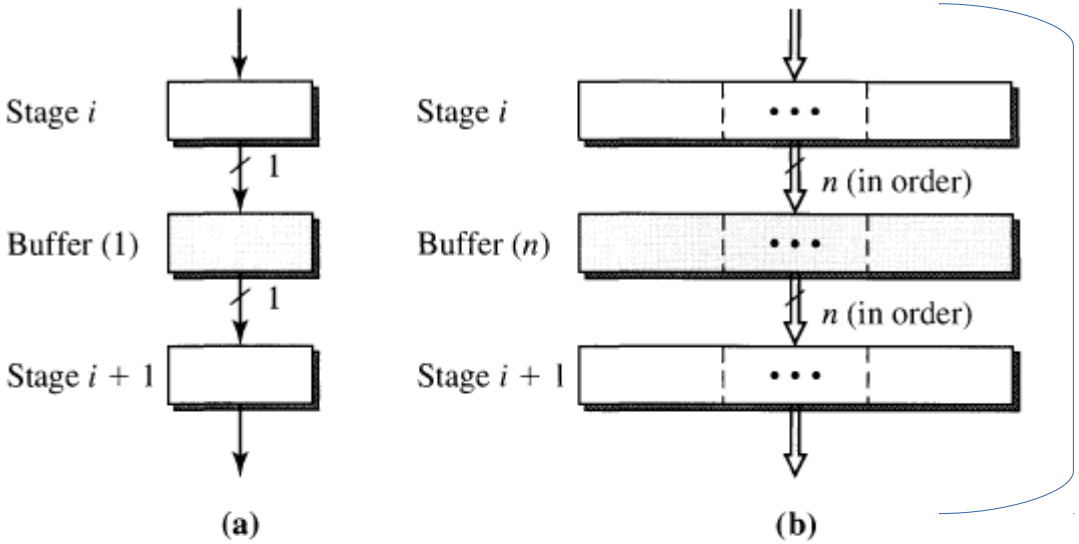
Computer System Architecture

Indian Institute of Technology Tirupati

jtt@iittp.ac.in

26th March, 2020

Superscalar Pipeline Stages

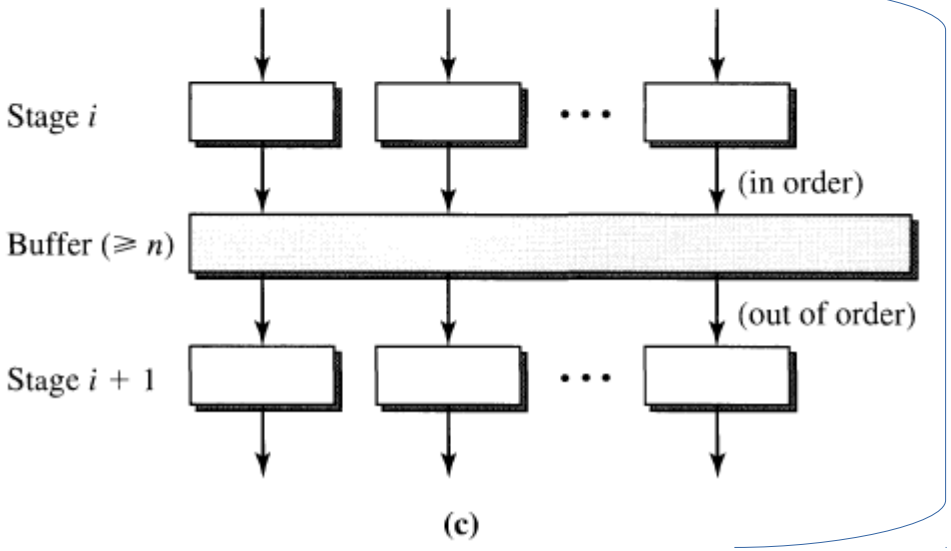


Progress in lock-up fashion

The issues:

- Control dependency
- Data dependency
- Need of program Order

Detection and Resolving



Progress in free fashion (dynamic fashion)

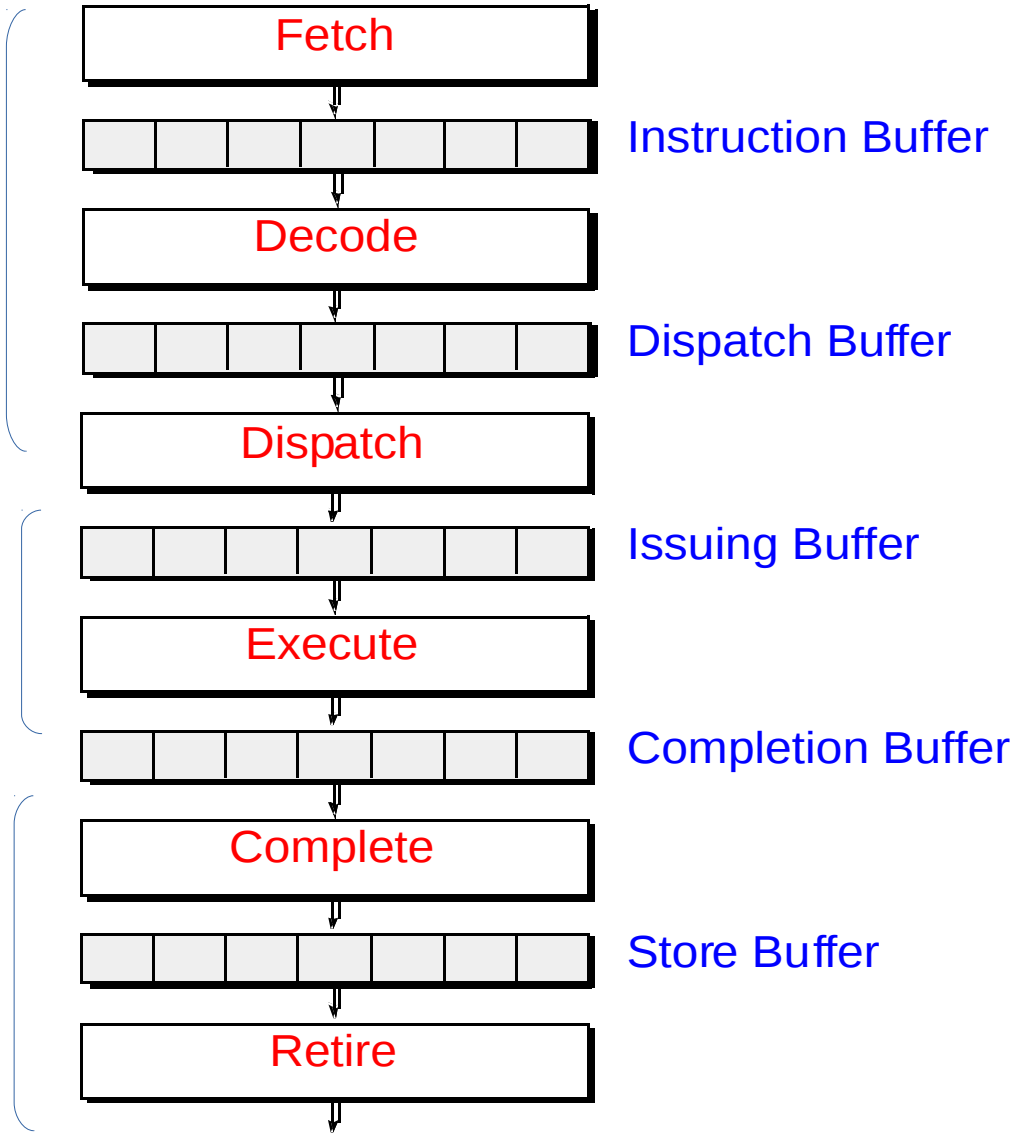
Superscalar Pipeline Stages

Dynamic superscalar pipeline

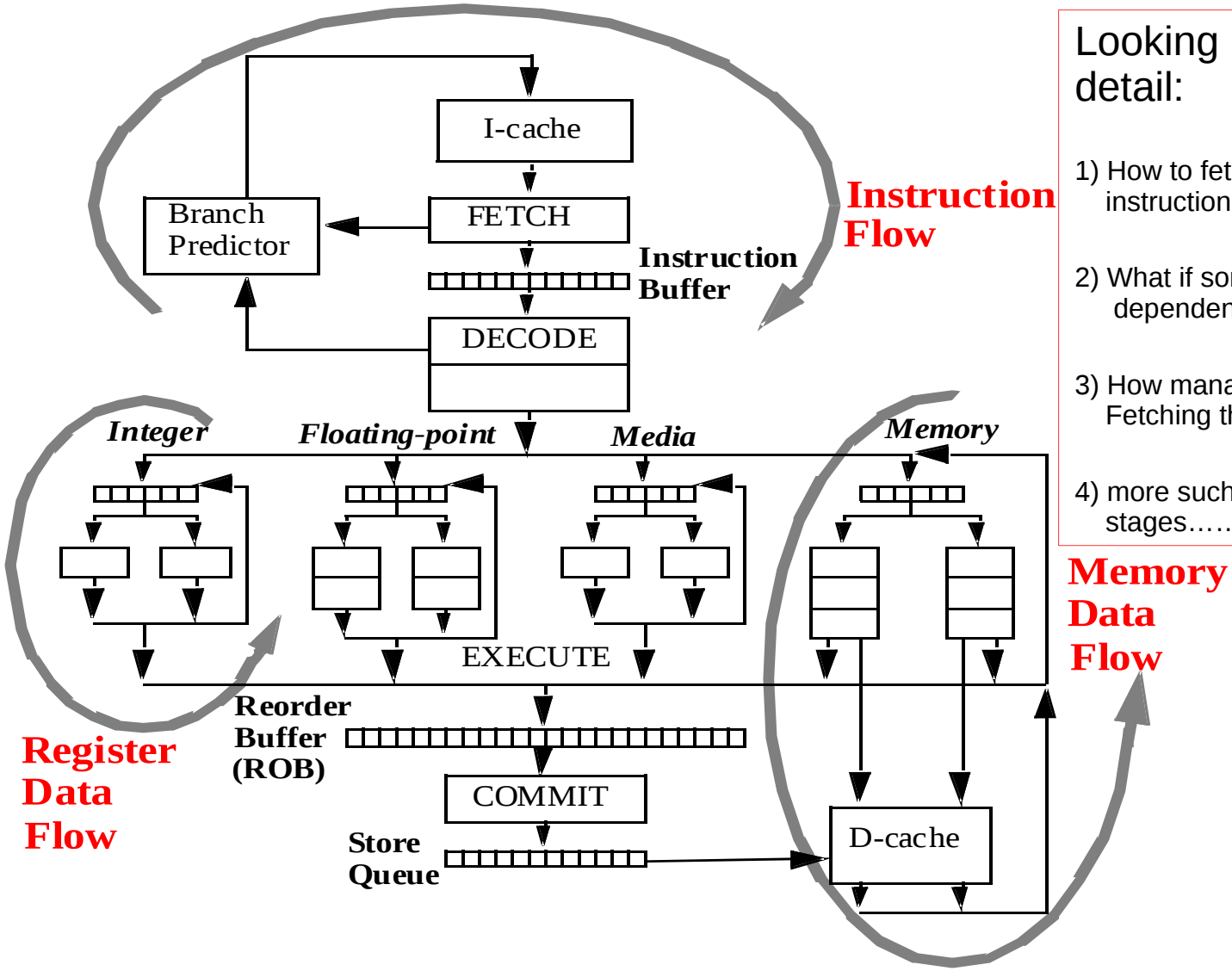
In program order

Out of order

In program order



Superscalar Challenges



Looking into the detail:

1) How to fetch N instruction simultaneously

2) What if some of them are dependent?

3) How manage memory while Fetching the instruction.

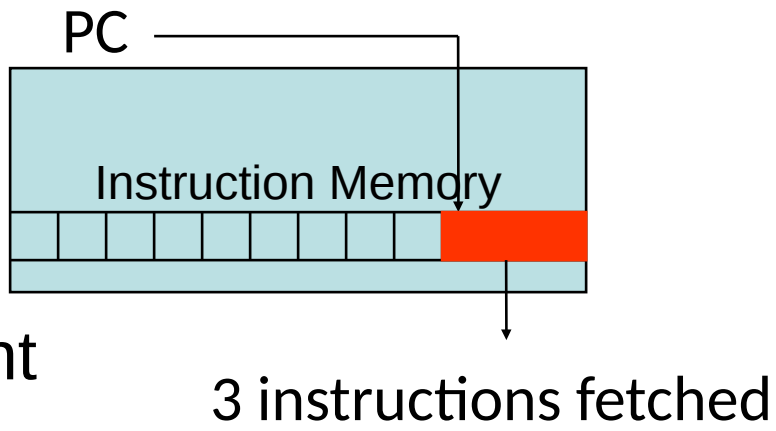
4) more such questions for each stages…….

Instruction Flow

Objective: to fetch *N* instructions per cycle for *N* width superscalar.

Challenges:

- Branch target misalignment
- Branches: control dependences
- Instruction cache misses



Instruction Flow

Objective: to fetch *N* instructions per cycle for *N* width superscalar.

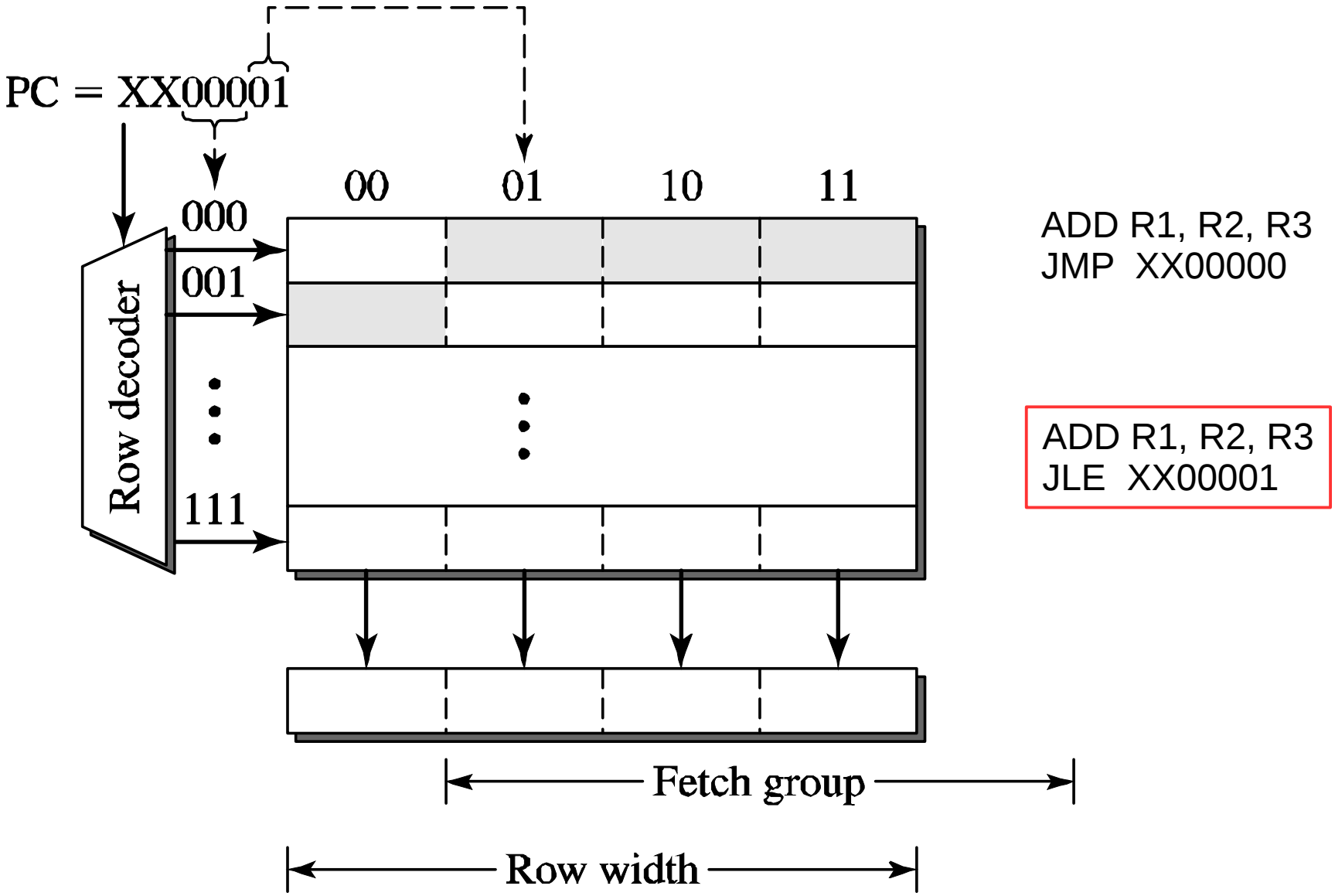
Challenges:

- Branch target misalignment
- Branches: control dependences
- Instruction cache misses

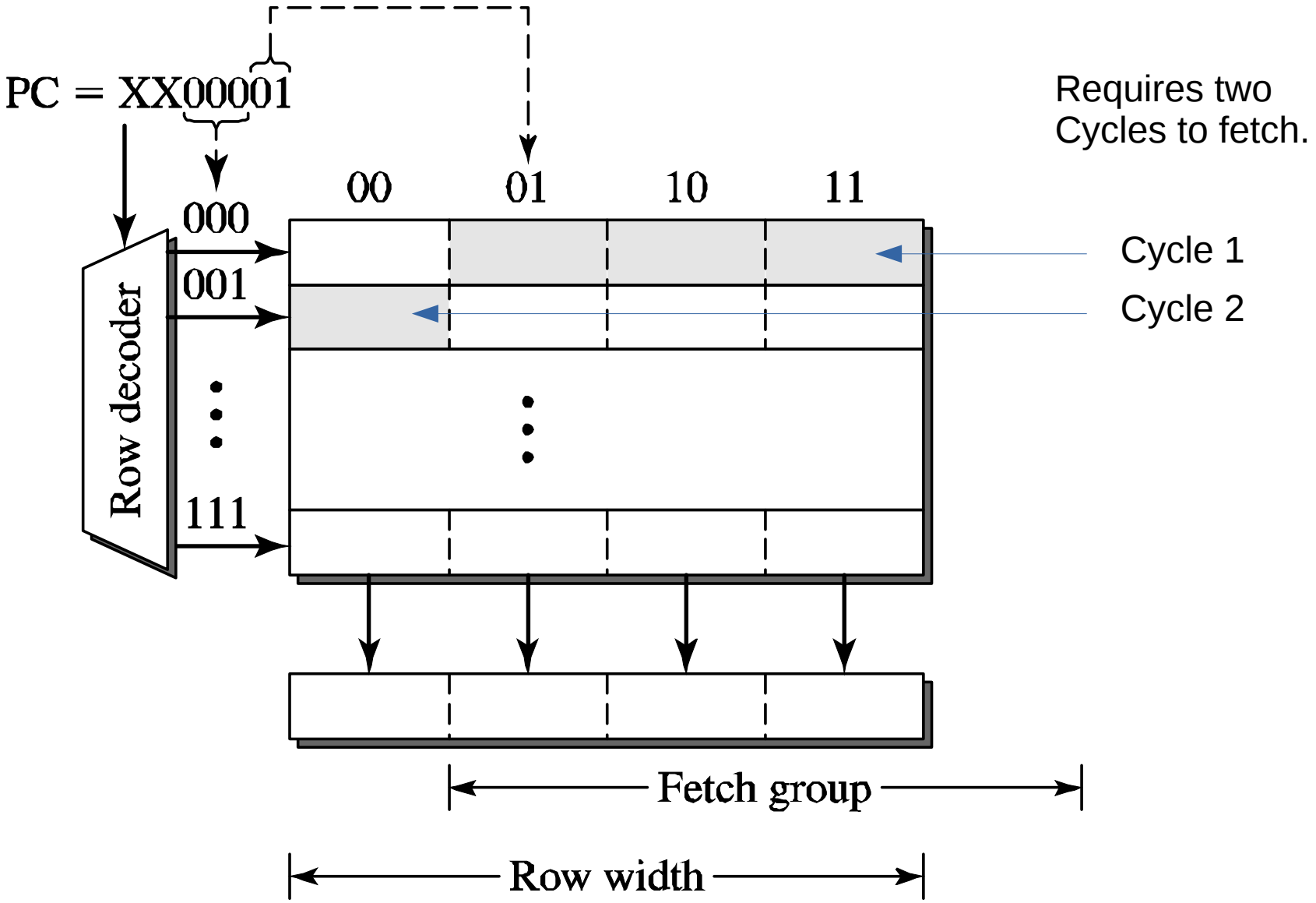
Solutions

- Code alignment (static vs. dynamic)
- Prediction/speculation
- Solution to cache misses

Fetch Alignment



Fetch Alignment



Fetch Alignment: Solutions

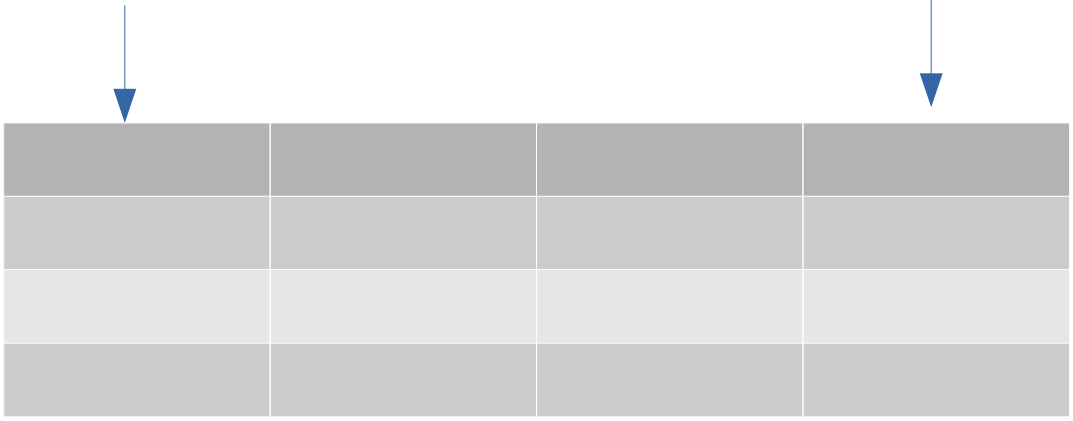
Solutions:

Compiler based **static alignment**
Hardware **dynamic alignment**

The static alignment:

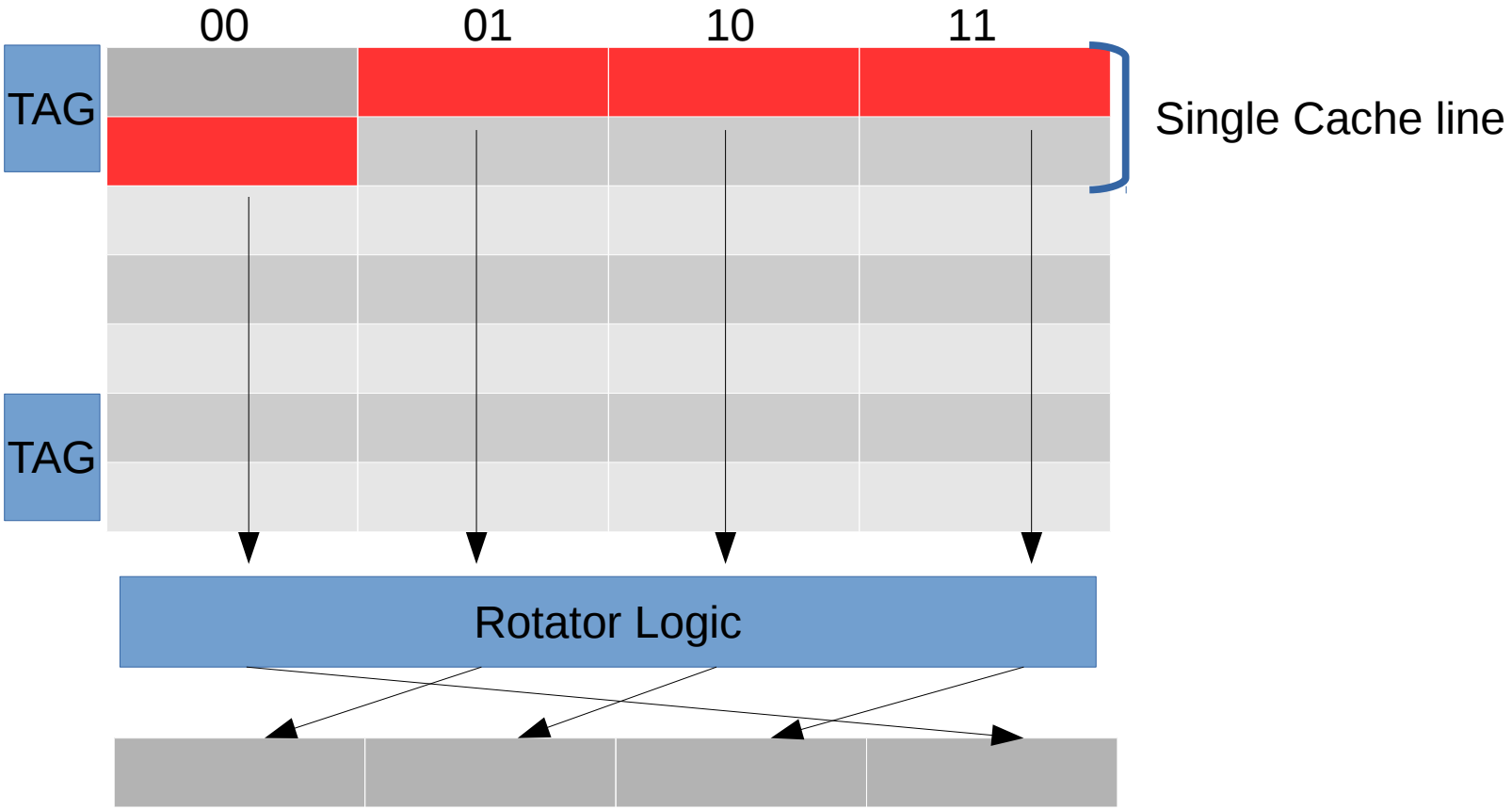
All the branch instructions
Must be placed here

All the target instructions should start from 0^{th} location



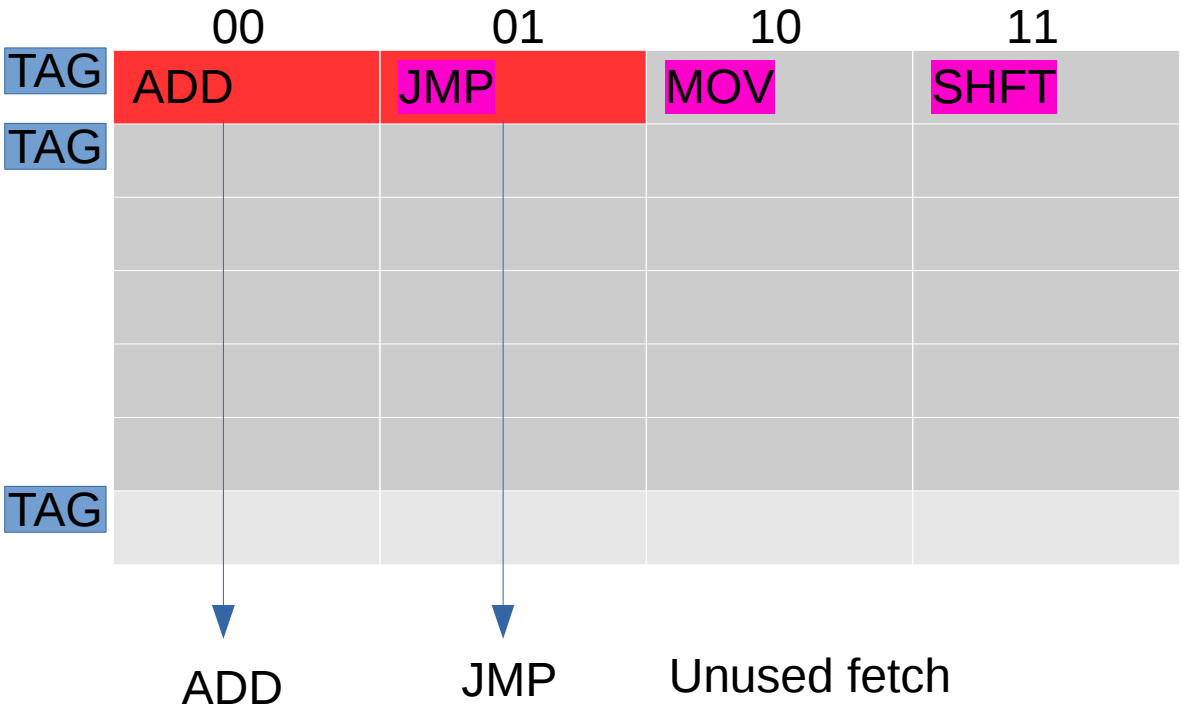
Fetch Alignment: Solutions

Size of fetch group < Cache line size



Fetch Limits Due to Branch

Size of fetch group < Cache line size



Instruction Fetch: IBM RS/6000

IBM RS/6000 Instruction cache architecture:

- 2 – way set associative I-Cache with a line size of 16 instructions (64 bytes)

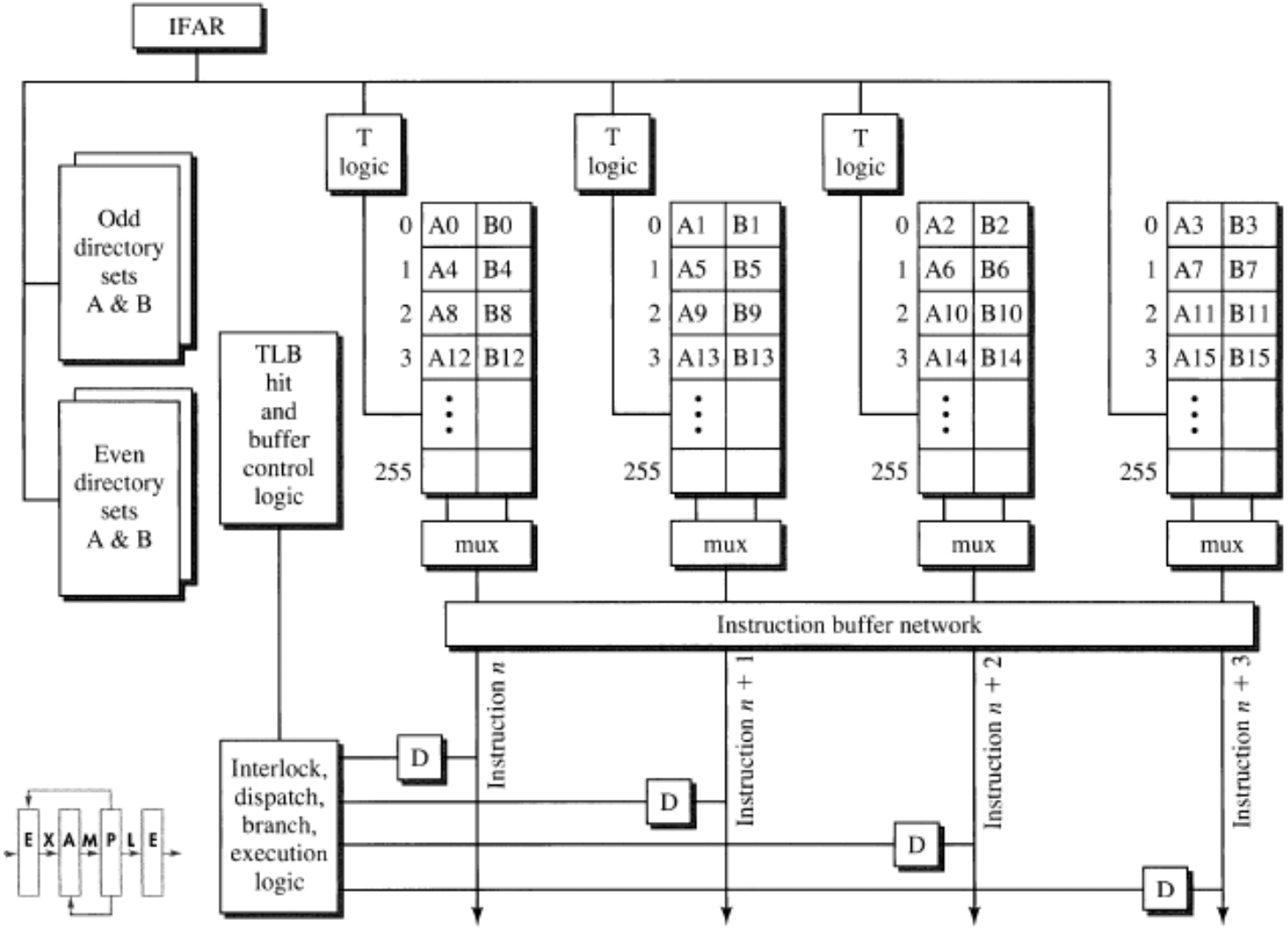
- Each row of the I-Cache stores 4 associative sets (two per set) of instructions

- Each line of I-cache spans four physical rows

- Physical I-cache array is actually composed of 4 independent sub-arrays

- One instruction can be accessed form one array

I-Cache Fetch Hardware: RS/6000



Two-away set associative I-Cache with auto-realignment

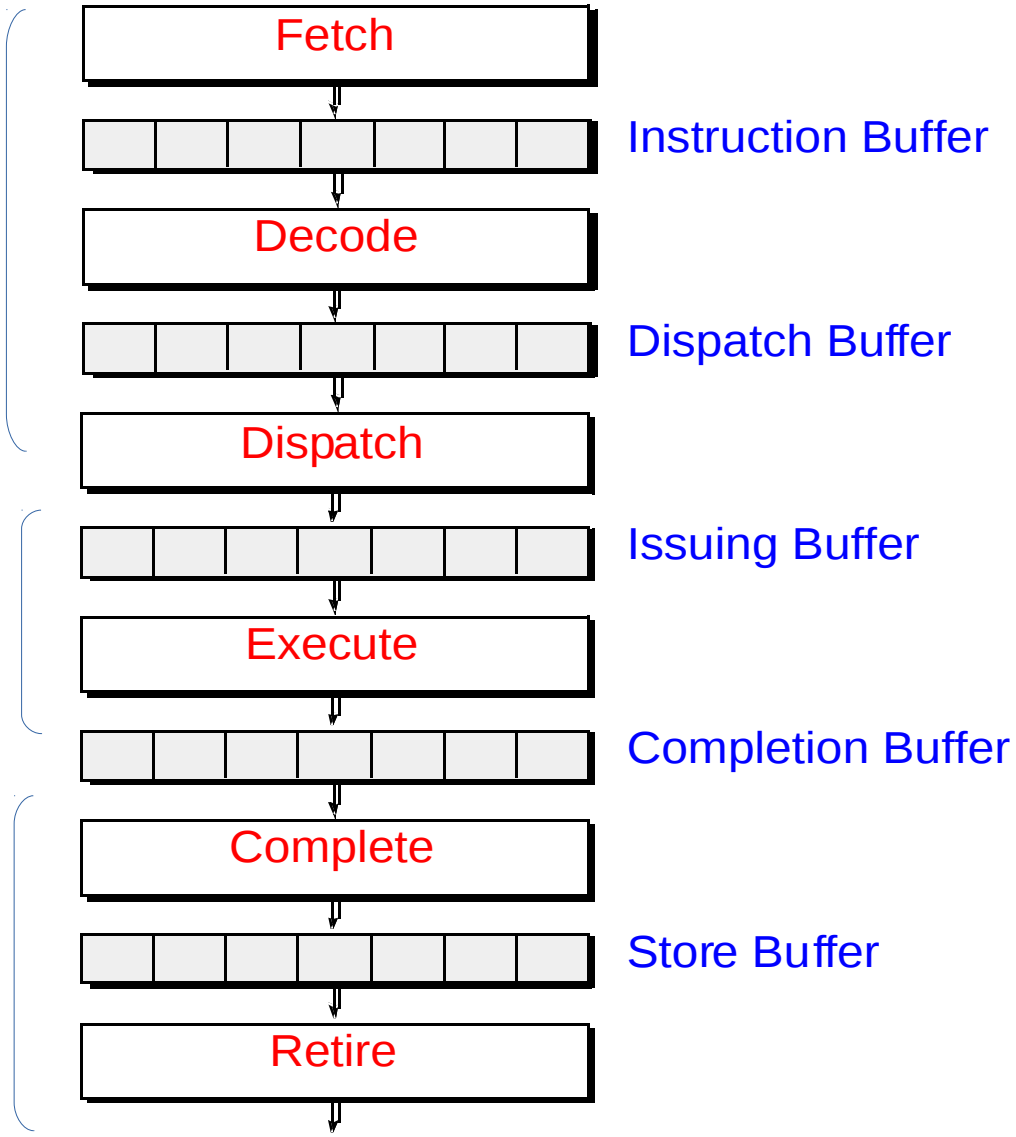
Superscalar Pipeline Stages

Dynamic superscalar pipeline

In program order

Out of order

In program order

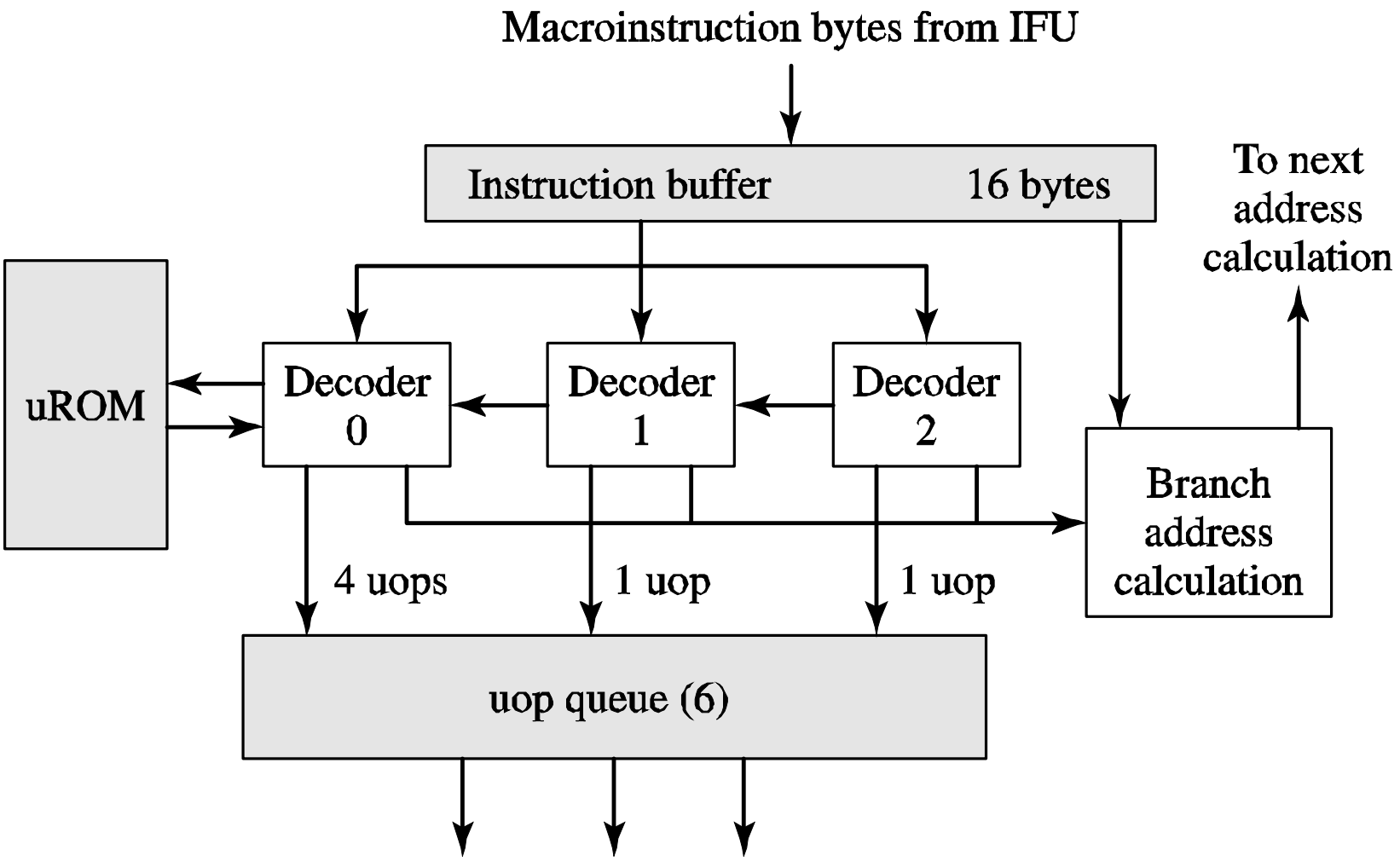


Challenges in Decoding

- Primary Tasks
  - Identify individual instructions (!)
  - Determine instruction types
  - Determine dependences between instructions

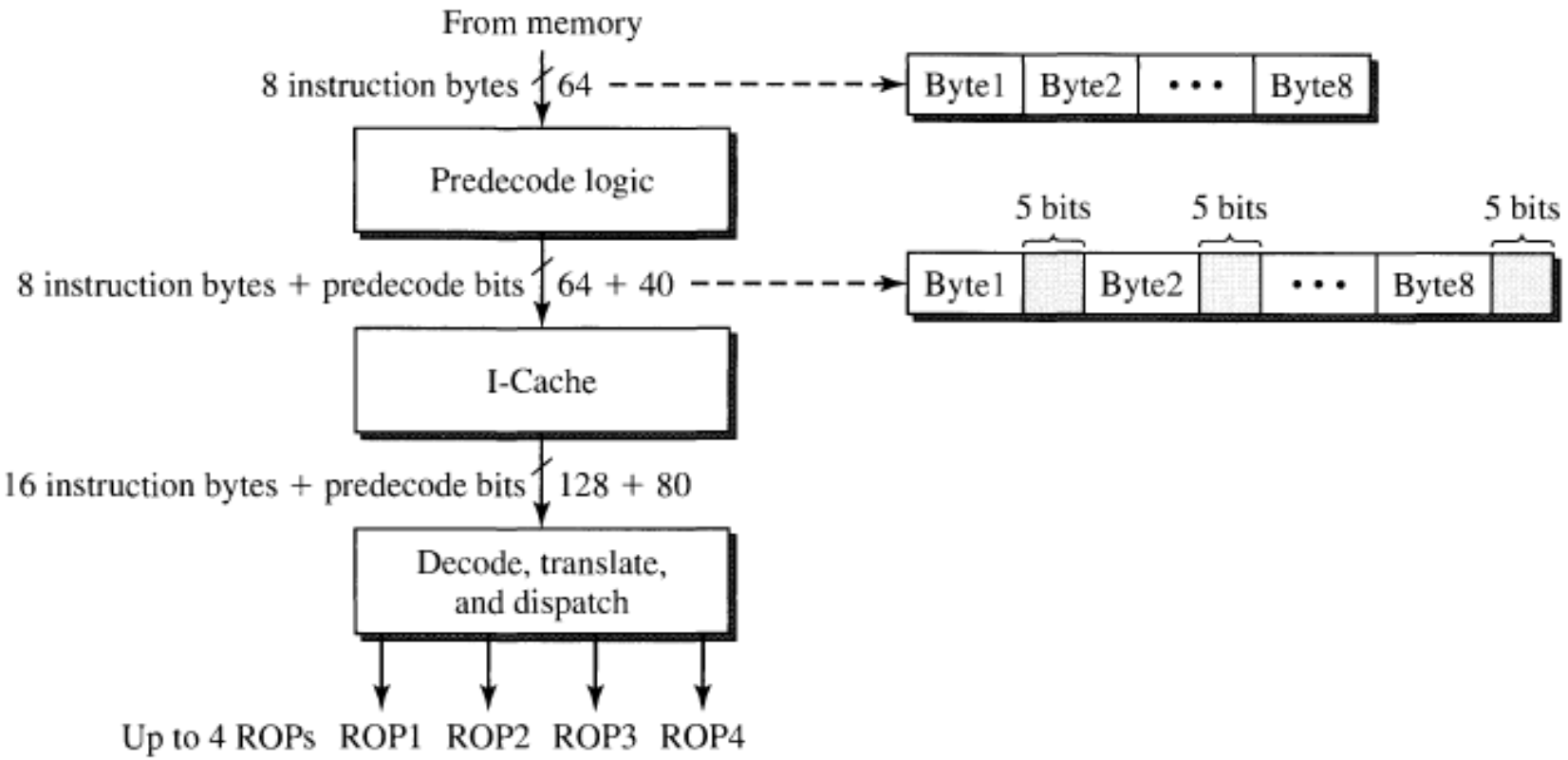
- Two important factors
  - Instruction set architecture
  - Pipeline width

Pentium Pro Fetch/Decode



Predecoding in AMD

- To simplify the decoding logic
- To avoid repeated decoding



Dependency Check and Resolve

The problem that would arise during operand fetch!

Solutions:

Score-board
Tomasulo's Algorithm
(dynamic scheduling)

Next lecture

Instruction Dispatching

IF | Decode | Operand Fetch | Dispatch | Other stage 1 | other stage 2 | ….

- Diversified pipeline: Multiple functional units
- Different type instructions executed by different Functional units
- Distributed control
- Operands are fetched from Register File
- Operands may not be ready yet
- Reservation station for instructions to be executed

Instruction Dispatch and Issue

- Parallel pipeline
  - Centralized instruction fetch
  - Centralized instruction decode
- Diversified pipeline (dynamic superscalar)
  - Distributed instruction execution

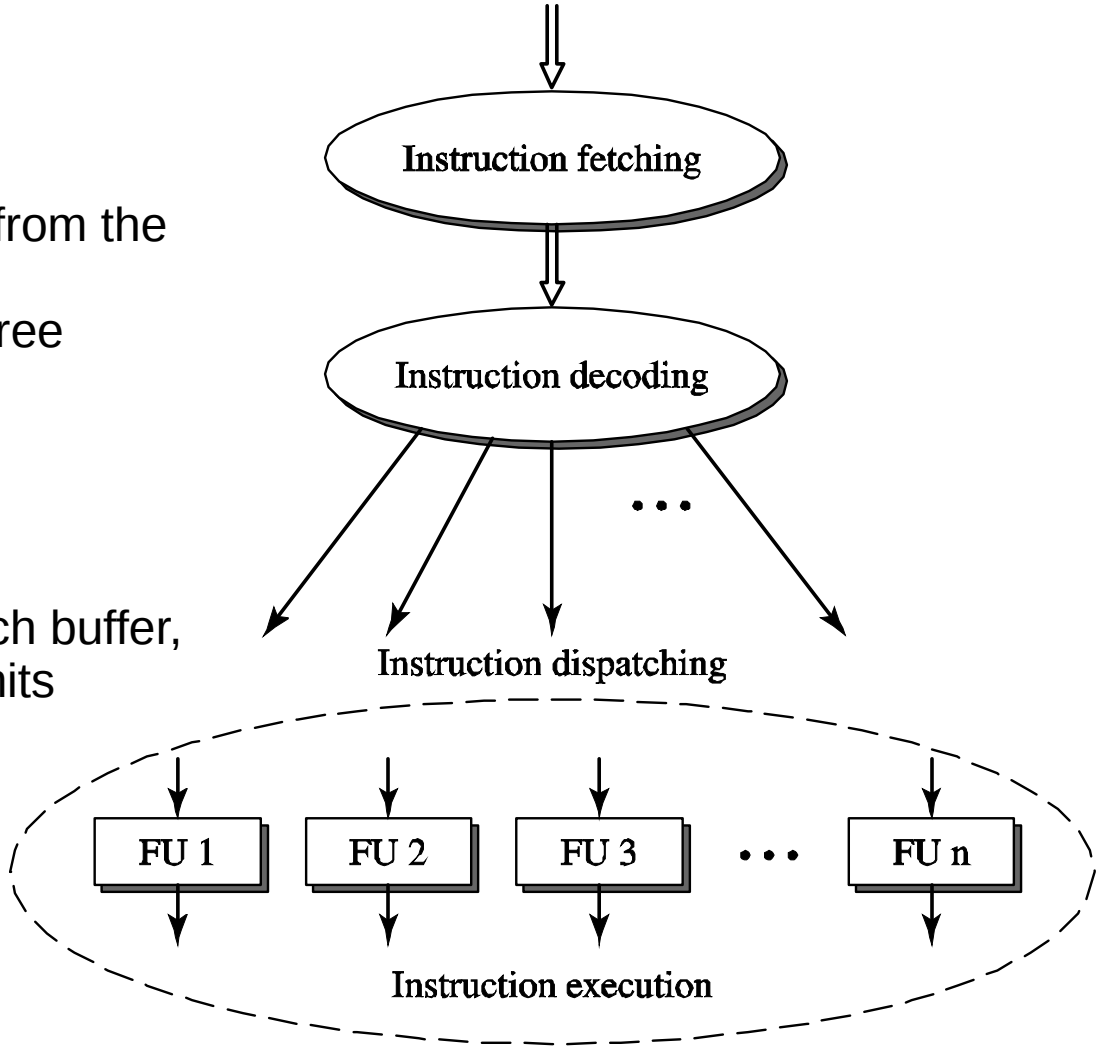
Necessity of Instruction Dispatch

Condition for Dispatch:

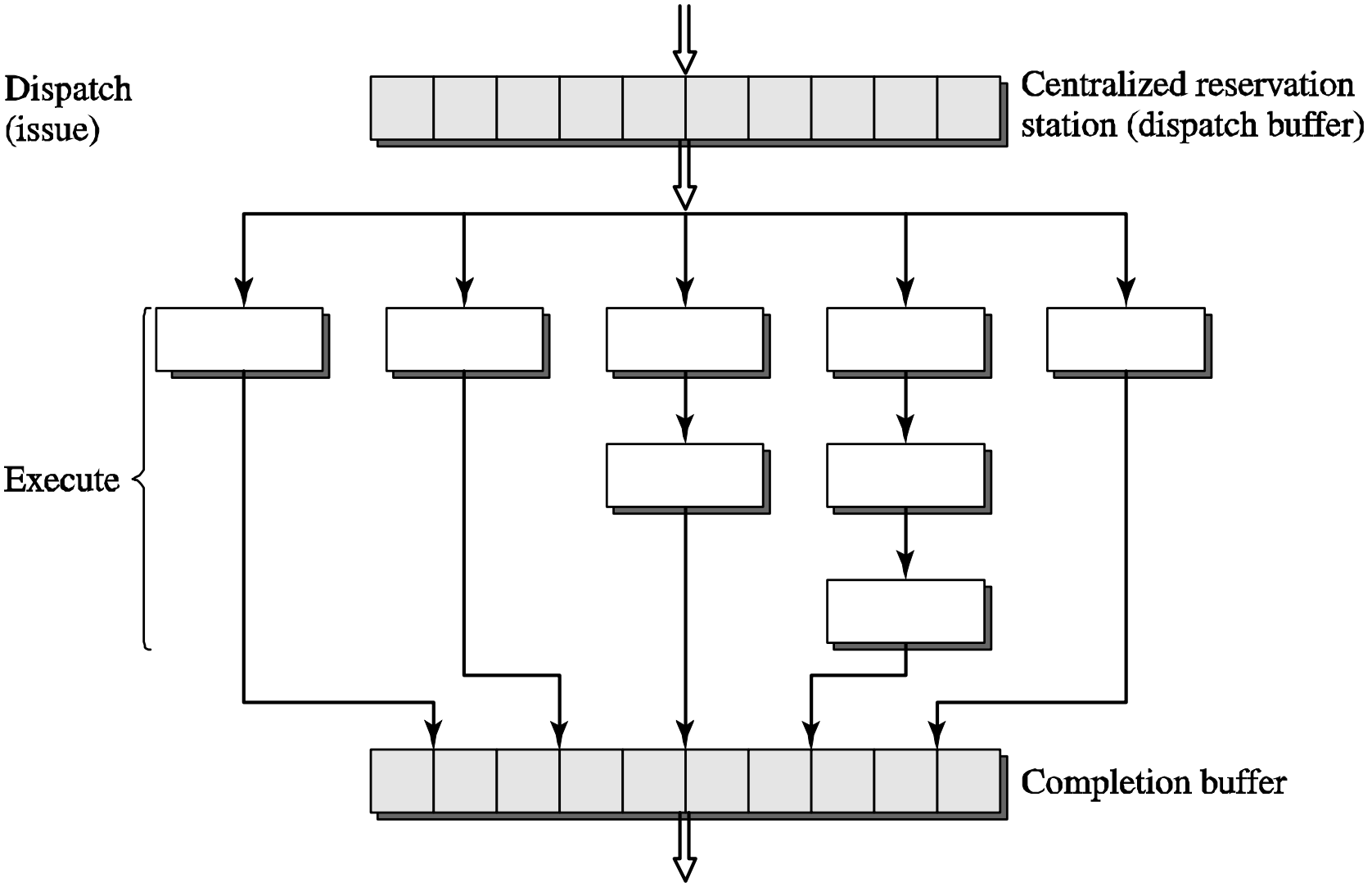
1) Decoding is completed
2) Registers data are read from the register file
3) If execution units is/are free

Takes instruction from Dispatch buffer, dispatch them to execution units

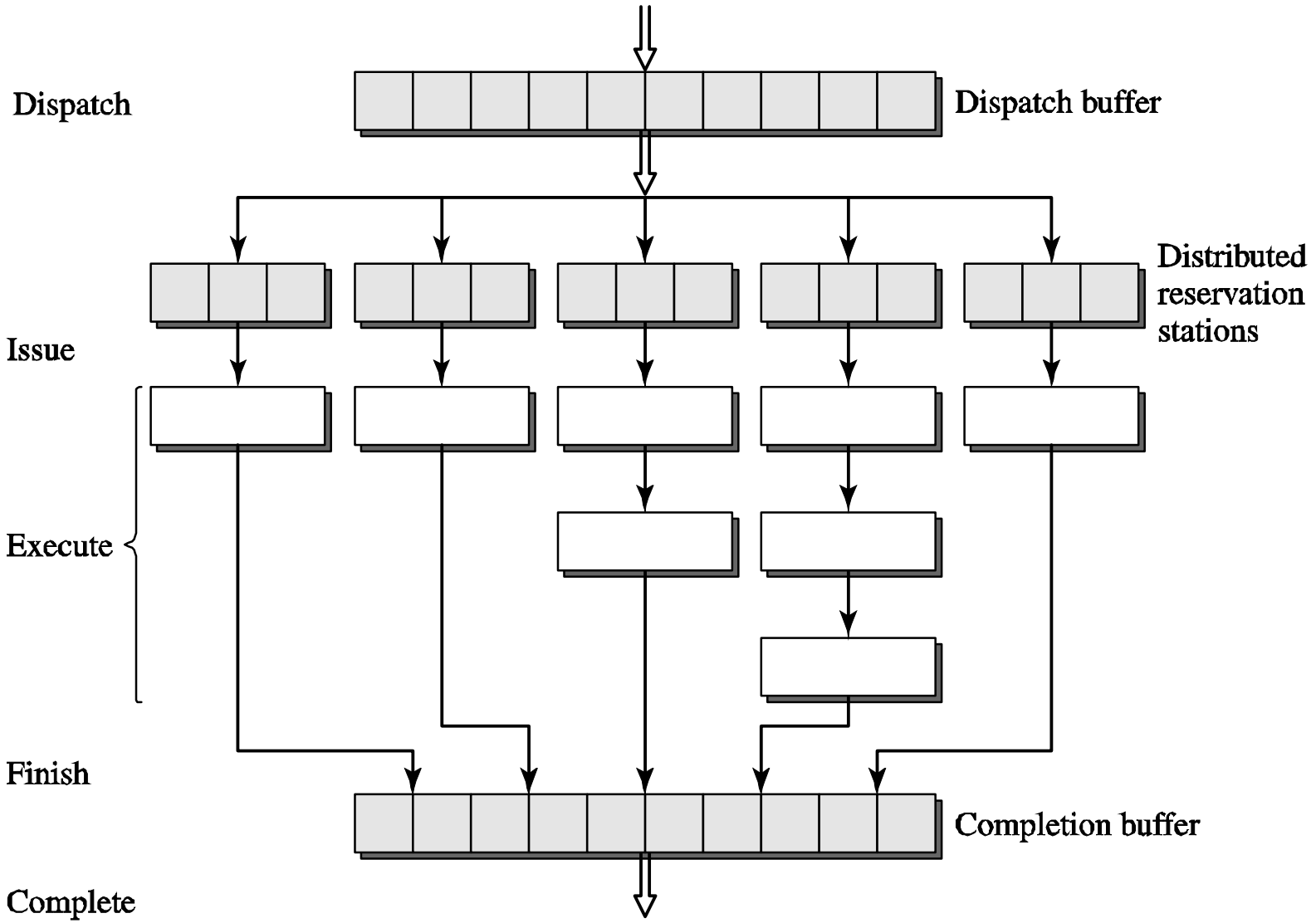
If execution units is/are not free?



Centralized Reservation Station



Distributed Reservation Station



Instruction Execution

The next challenges is to execute the dispatched instructions, objective is performance improvement.

- How to decide the size of reservation station?
- What type of execution units are necessary?
- How many execution of each type is needed?
- How many branch unit is desirable?
- How many load/store unit is desirable?
- How many ALU is needed
- How many FP unit is needed?

Why these are such a big issues? Think of program diversity!

Instruction Execution

The answer:

- Workload needs to be analyzed

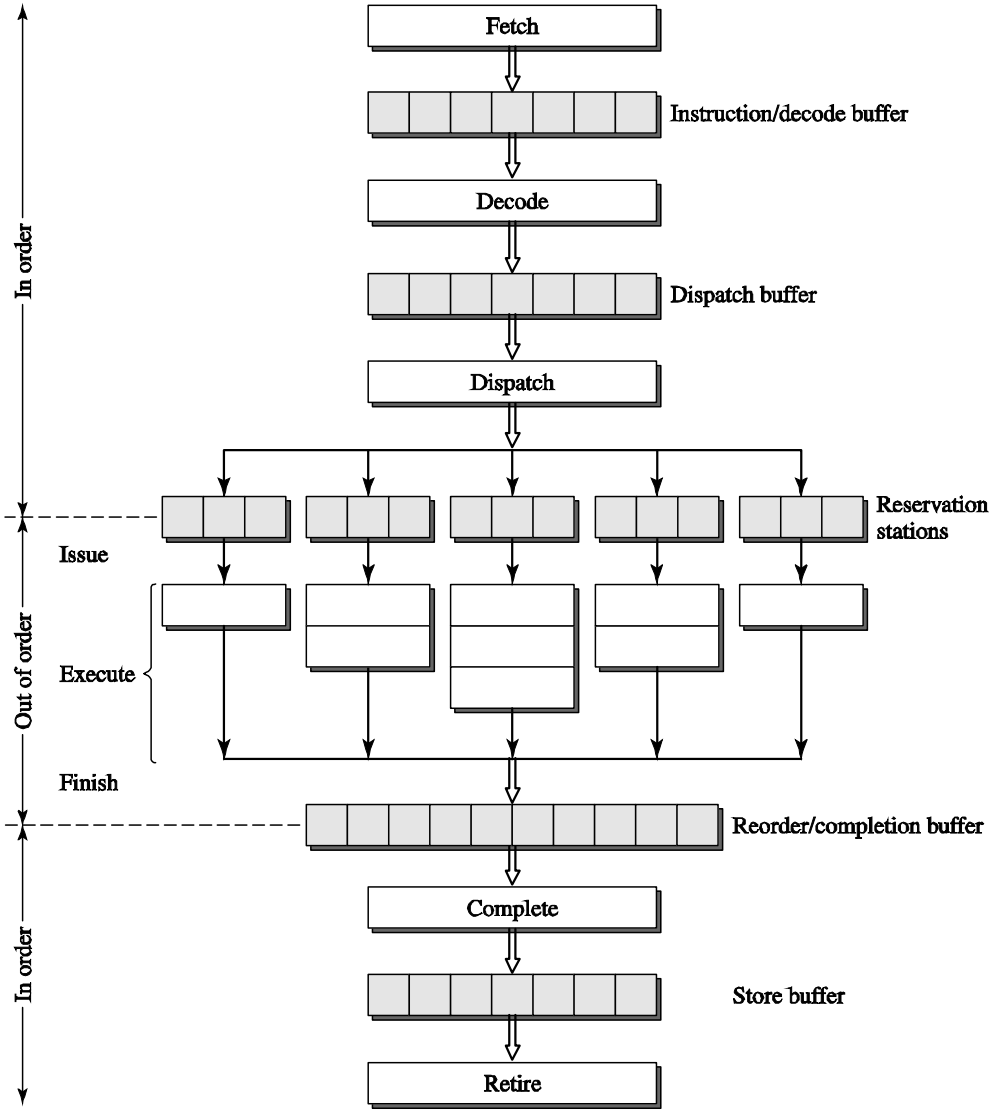
- The port numbers of Register file and D-Cache Matters.

- Wherever there is limitation in ports, the execution needs to be serialized

- Inter-functional unit data sharing network (for data forwarding)

Completion/Retirement

- Out-of-order execution
  - ALU instructions
  - Load/store instructions
- In-order completion/retirement
  - Precise exceptions
  - Memory coherence and consistency
- Solutions
  - Reorder buffer
  - Store buffer



Challenges in Superscalar Processor

How to maintain through-put of *n* for n-wide super-scalar?

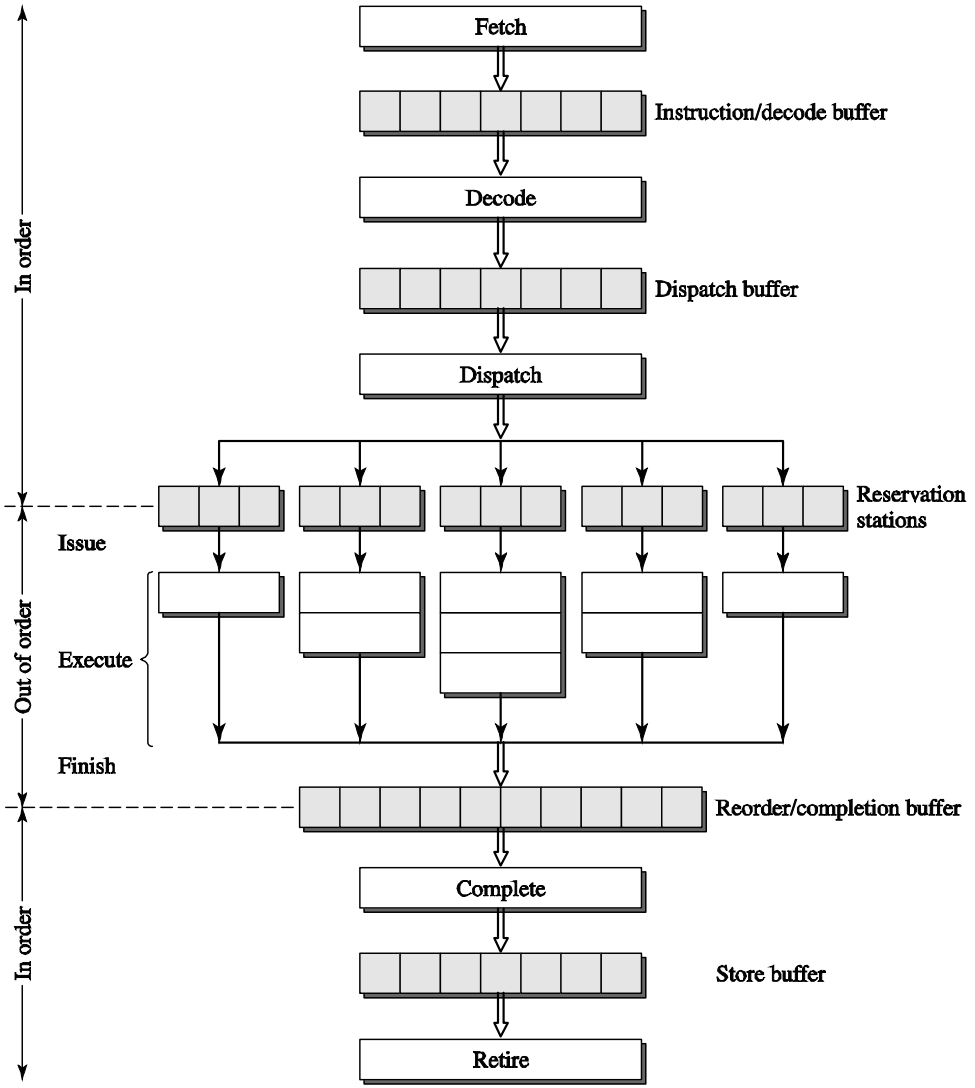
Completing *n* instructions per cycle

Recall the limits of scalar pipeline:

IPC <= 1

Superscal goal:

IPC ~= n

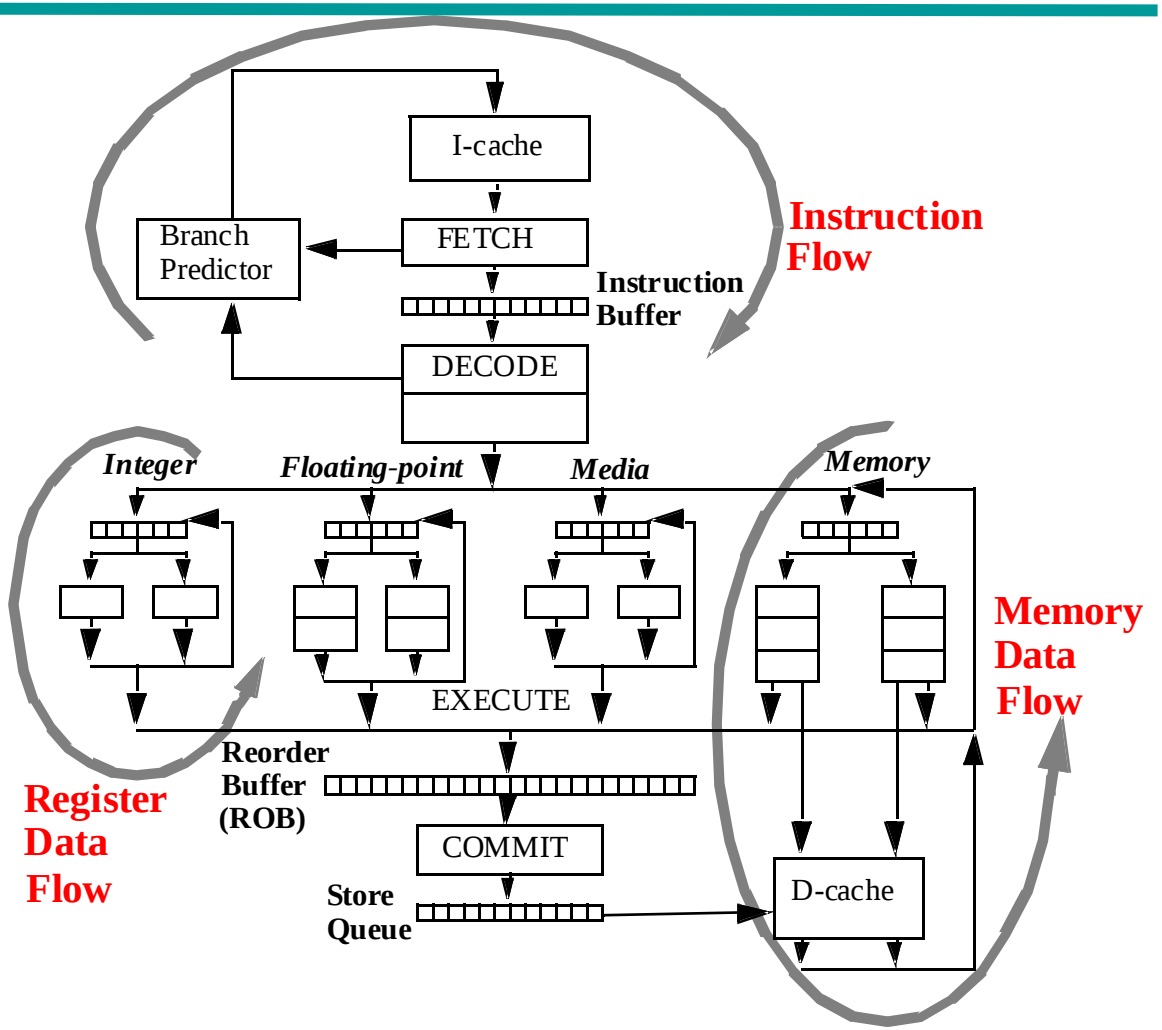


Superscalar Techniques

- The ultimate performance goal of a superscalar pipeline is to achieve maximum throughput of instruction processing
- Instruction processing involves
  - Instruction flow – Branch instructions
  - Register data flow – ALU instructions
  - Memory data flow – L/S instructions
- Max Throughput – Min (Branch, ALU, Load penalty)

Superscalar Challenges

Branch instructions and Control-flow path



Register Data dependency.

Primarily to deal with Register file

Memory data dependency
Mainly to deal with D-cache

Instruction Flow

- Goal of Instruction Flow
  - Supply processor with maximum number of useful instructions every clock cycle
  - To be able to fetch ***n*** instructions every cycle
- Impediments
  - Branches and jumps
  - Finite I-Cache
    - Capacity (on cache miss, the fetch has to stall)
    - Bandwidth restrictions

      (how many instruction can be read in a single clock cycle)

Branch Types and Implementation

1. Types of Branches
   A. Conditional or Unconditional
   B. Save PC?
   C. How is target computed?
      - Single target (immediate, PC + immediate)
      - Multiple targets (register)
2. Branch Architectures
   A. Condition code or condition registers
   B. Register

Branches – PowerPC

Example of branch instructions from PowerPC

Branch (unconditional, no save PC, PC+imm)
Branch absolute (uncond, no save PC, imm)
Branch and link (uncond, save PC, PC+imm)
Branch abs and link (uncond, save PC, imm)
Branch conditional (conditional, no save PC, PC+imm)
Branch cond abs (cond, no save PC, imm)
Branch cond and link (cond, save PC, PC+imm)
Branch cond abs and link (cond, save PC, imm)
Branch cond to link register (cond, don't save PC, reg)
Branch cond to link reg and link (cond, save PC, reg)
Branch cond to count reg (cond, don't save PC, reg)
Branch cond to count reg and link (cond, save PC, reg)

Branches – DEC Alpha

Alpha: 3 Types of Branches

Conditional branch (cond, no save PC, PC+imm)

Bxx Ra, disp

Unconditional branch (uncond, Save PC, PC+imm)

Br Ra, disp

Jumps (uncond, save PC, Register)

J Ra

Branches – MIPS

MIPS Instruction Set

6 Types of Branches

Jump (uncond, no save PC, imm)

Jump and link (uncond, save PC, imm)

Jump register (uncond, no save PC, register)

Jump and link register (uncond, save PC, register)

Branch (conditional, no save PC, PC+imm)

Branch and link (conditional, save PC, PC+imm)

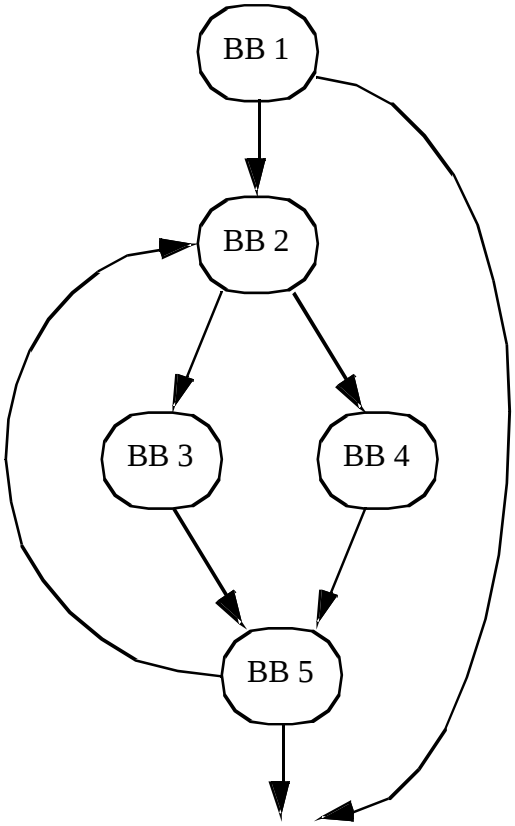
Difficulty of Branch Instruction

They very severely affects the performance primarily due to diversified program flow.

Effects of Branches

- Fragmentation of I-Cache lines
- Need to determine branch direction
- Need to determine branch target
- Use up execution resources
  - Pipeline drain/fill

Control Flow of a Program



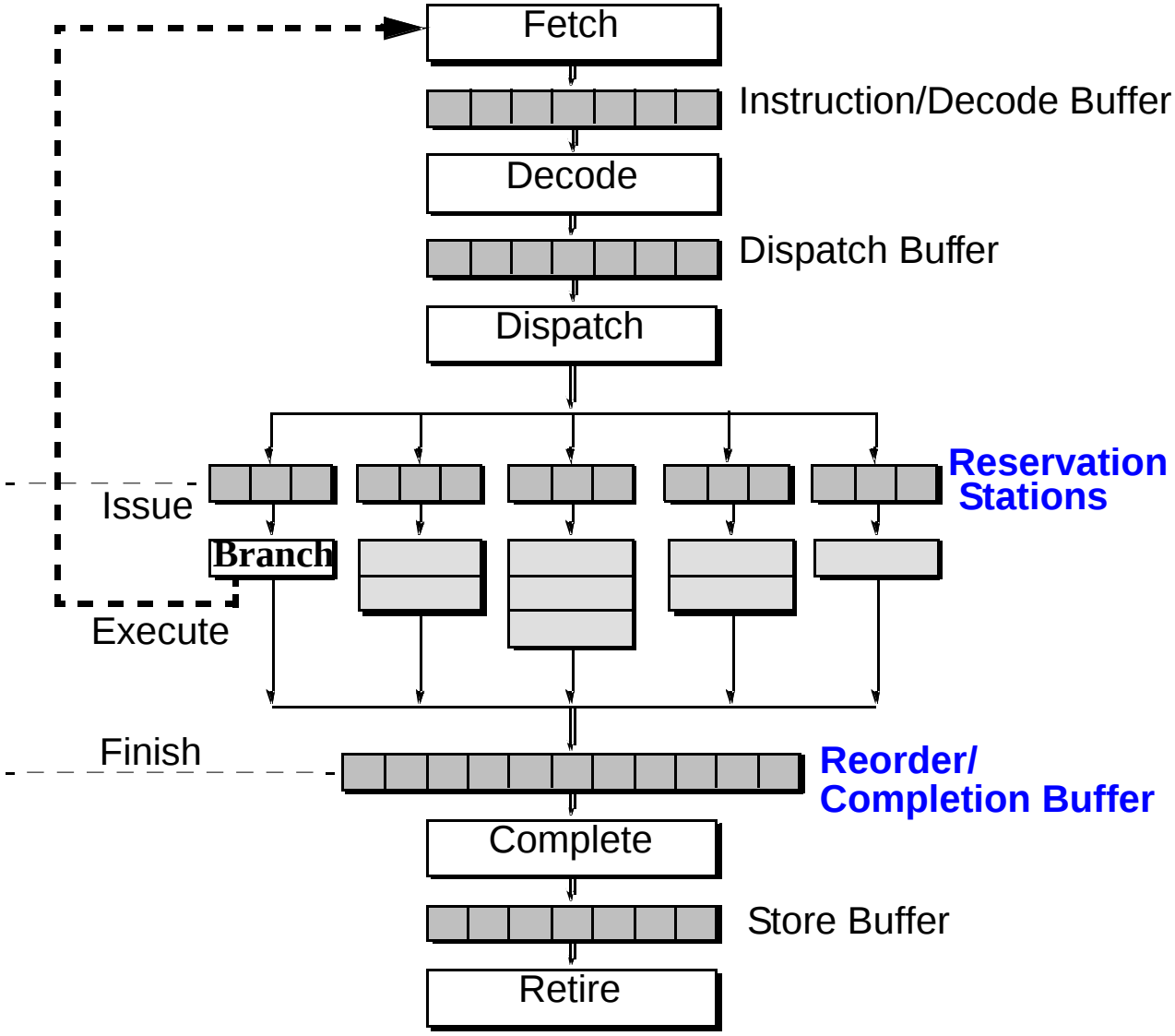

```
main:
      addi r2, r0, A
      addi r3, r0, B
      addi r4, r0, C          BB 1
      addi r5, r0, N
      add  r10,r0, r0
      bge  r10,r5, end
loop:
      lw   r20, 0(r2)
      lw   r21, 0(r3)         BB 2
      bge  r20,r21,T1
      sw   r21, 0(r4)         BB 3
      b    T2
T1:
      sw   r20, 0(r4)         BB 4
T2:
      addi r10,r10,1
      addi r2, r2, 4
      addi r3, r3, 4          BB 5
      addi r4, r4, 4
      blt  r10,r5, loop
end:
```

- Control Flow Graph
  - Shows possible paths of control flow through basic blocks (BB)

Program Control Flow

- Implicit Sequential Control Flow
  - Static Program Representation
    - Control Flow Graph (CFG)
    - Nodes = basic blocks
    - Edges = Control flow transfers
  - Physical Program Layout
    - Mapping of CFG to linear program memory
    - Implied sequential control flow
  - Dynamic Program Execution
    - Traversal of the CFG nodes and edges (e.g. loops)
    - Traversal dictated by branch conditions
  - Dynamic Control Flow
    - Deviates from sequential control flow
    - Disrupts sequential fetching
    - Can stall IF stage and reduce I-fetch bandwidth

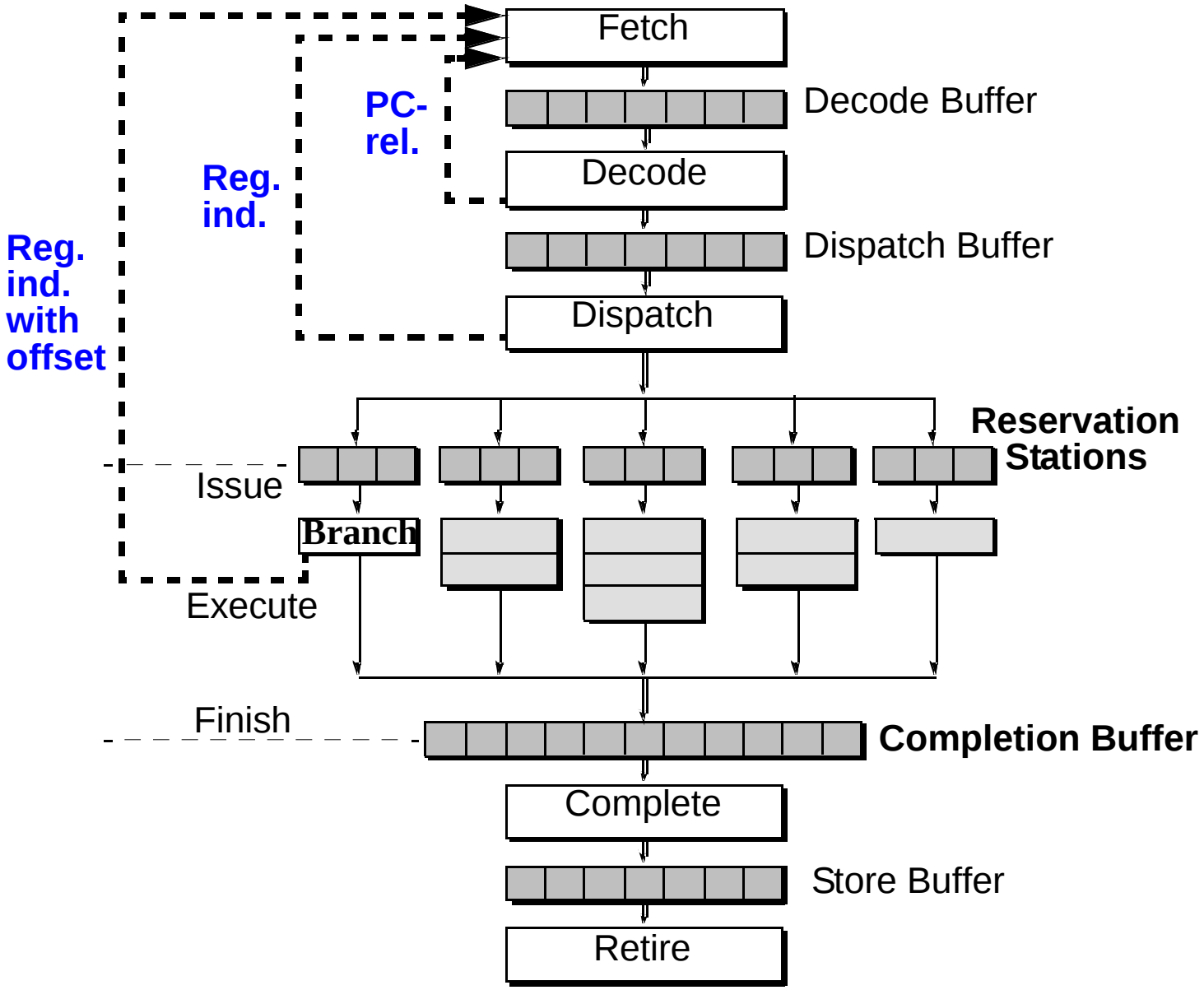
Disruption of Sequential Control Flow



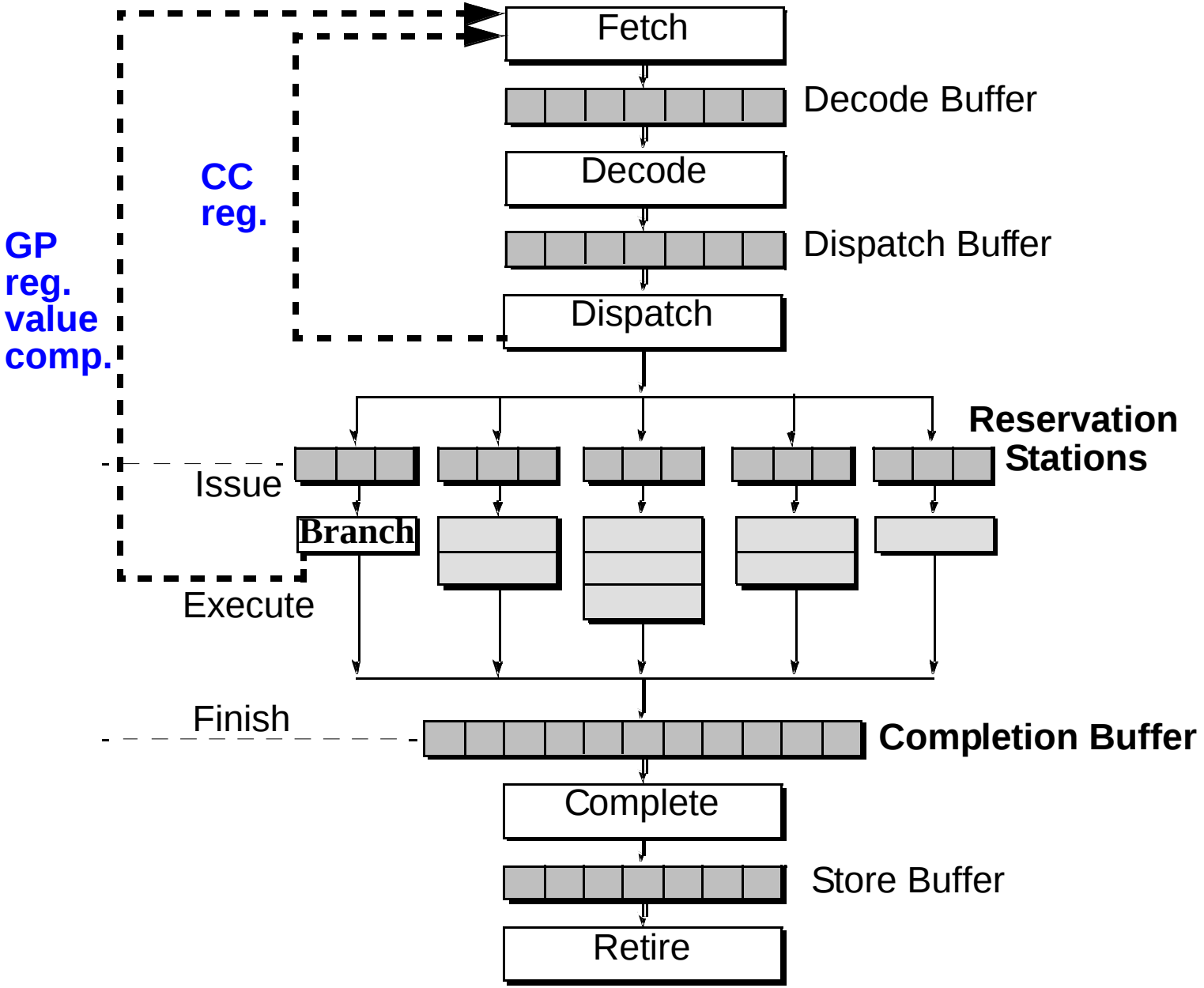
Branch Prediction

- Condition resolution | Condition speculation
  - Access register:
    - Condition code register, General purpose register
  - Perform calculation:
    - Comparison of data register(s)
- Target address generation | Target Speculation
  - Access register:
    - PC, General purpose register, Link register
  - Perform calculation:
    - +/- offset, autoincrement, autodecrement

Target Address Generation



Condition Resolution



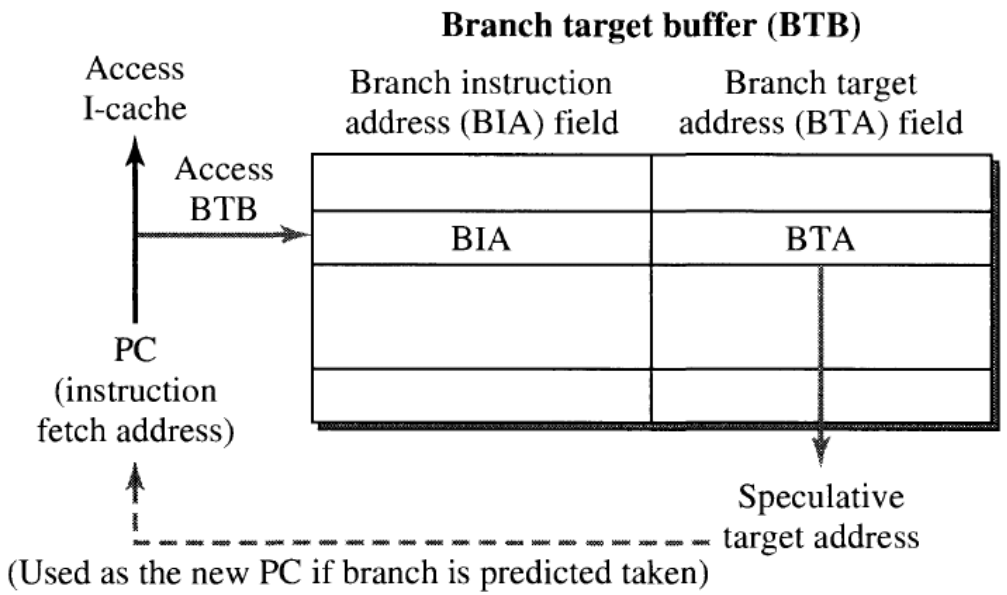
Solving Target Address Generation

Can I predict the target address during the fetch stage itself?

This would solve (delayed) address generation penalty!

HOW?

Branch/Jump Target Prediction

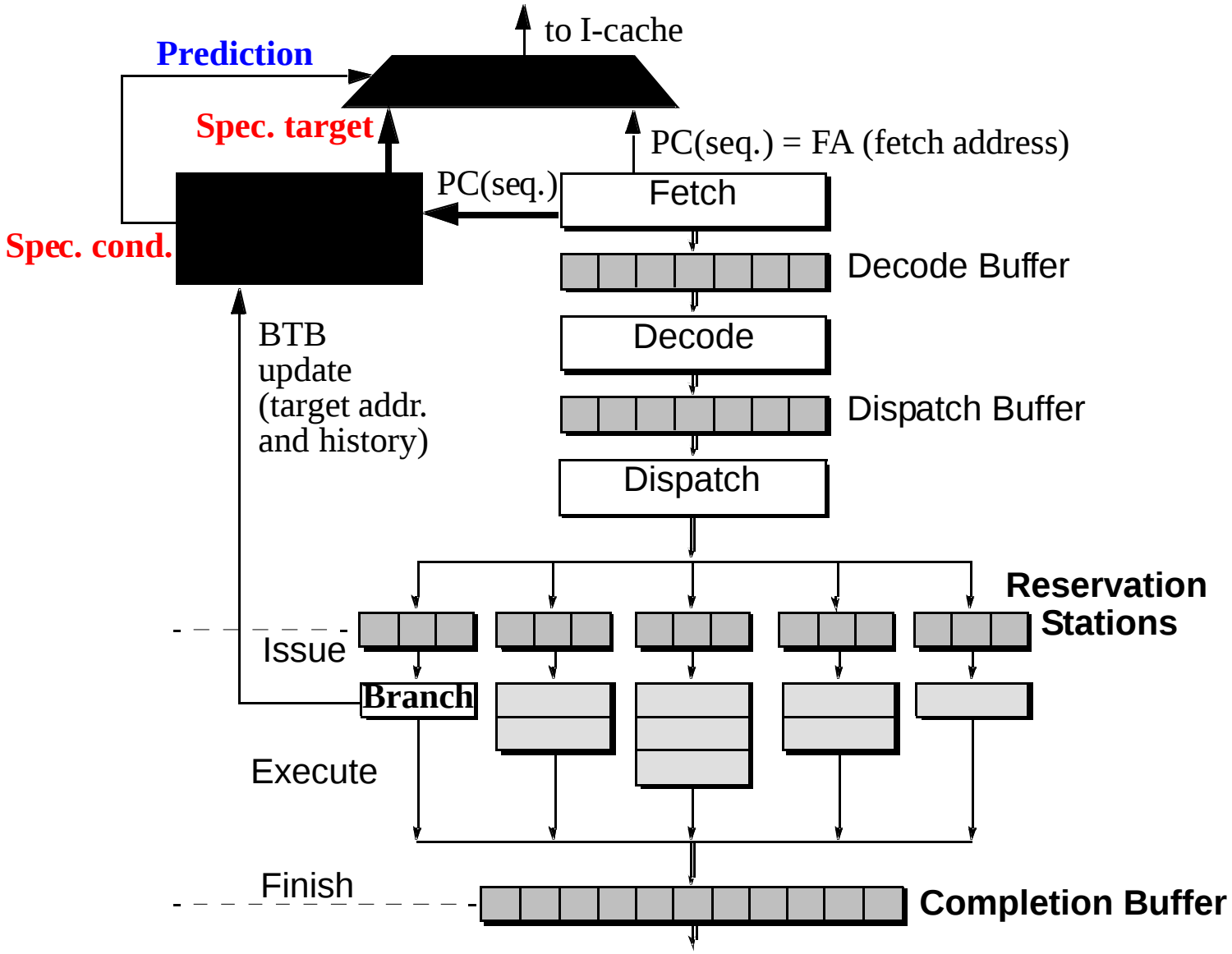


- Branch Target Buffer: small cache in fetch stage
  - Previously executed branches, address, taken history, target(s)
- Fetch stage compares current FA against BTB
  - If match, use prediction
  - If predict taken, use BTB target
- When branch executes, BTB is updated
- Optimization:
  - Size of BTB: increases hit rate
  - Prediction algorithm: increase accuracy of prediction

Dynamic Branch Prediction

- Main advantages:
  - Learn branch behavior autonomously
    - No compiler analysis, heuristics, or profiling
  - Adapt to changing branch behavior
    - Program phase changes branch behavior
- First proposed in 1979-1980
  - US Patent #4,370,711, Branch predictor using random access memory, James. E. Smith
- Continually refined since then

Branch Instruction Speculation

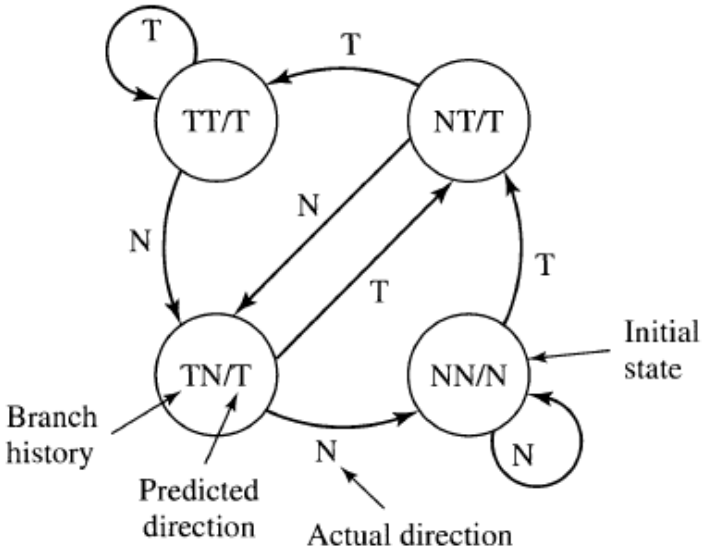


Branch Instruction Speculation

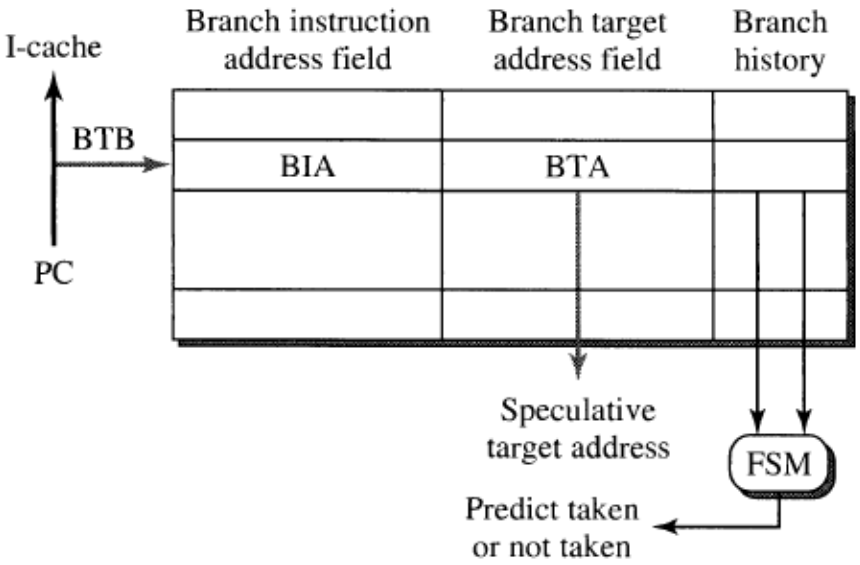
- Simplest predictor:
  - just predict not taken
  - 1-bit predictor
  - 2-bit predictor
- Advance predictors:
  - Correlating predictor
  - Tournament predictor

Branch Instruction Speculation

Implementation of 2 -bit predictor and extension to advance predictor.



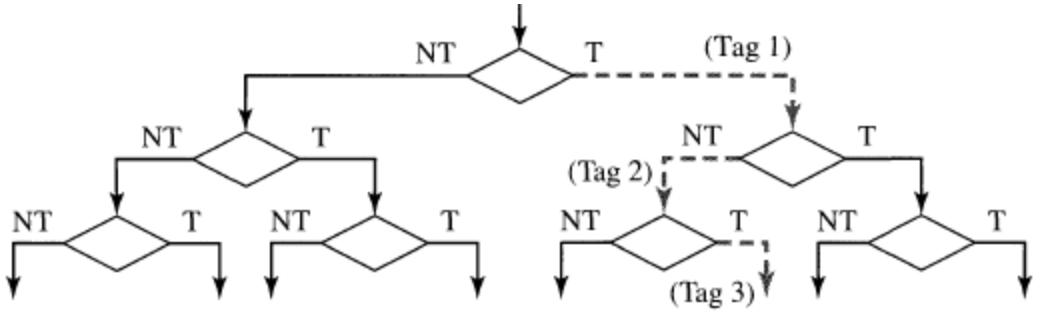
State-transition diagram



Implementation using BTB

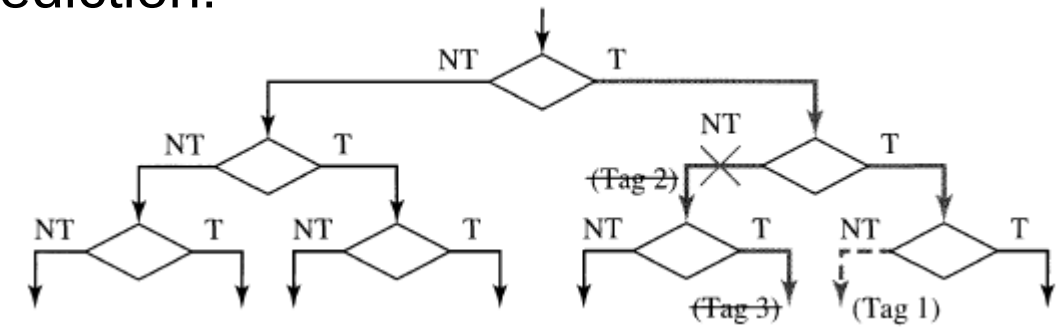
Recovery from Misprediction

The predicted path:



The important thing is the TAG

Recovery from wrong prediction:



Branch Instruction Speculation

- Simplest predictor: Just predict **not taken**
- 1-bit predictor
- 2-bit predictor

- Advance predictors:
  - Correlating predictor
    - Tournament predictor

Next Lecture: Prediction