

# Out-of-order (OoO) Processing

Suvinay Subramanian

(adapted from prior 6.823 offerings,  
ack: Nathan Beckmann)

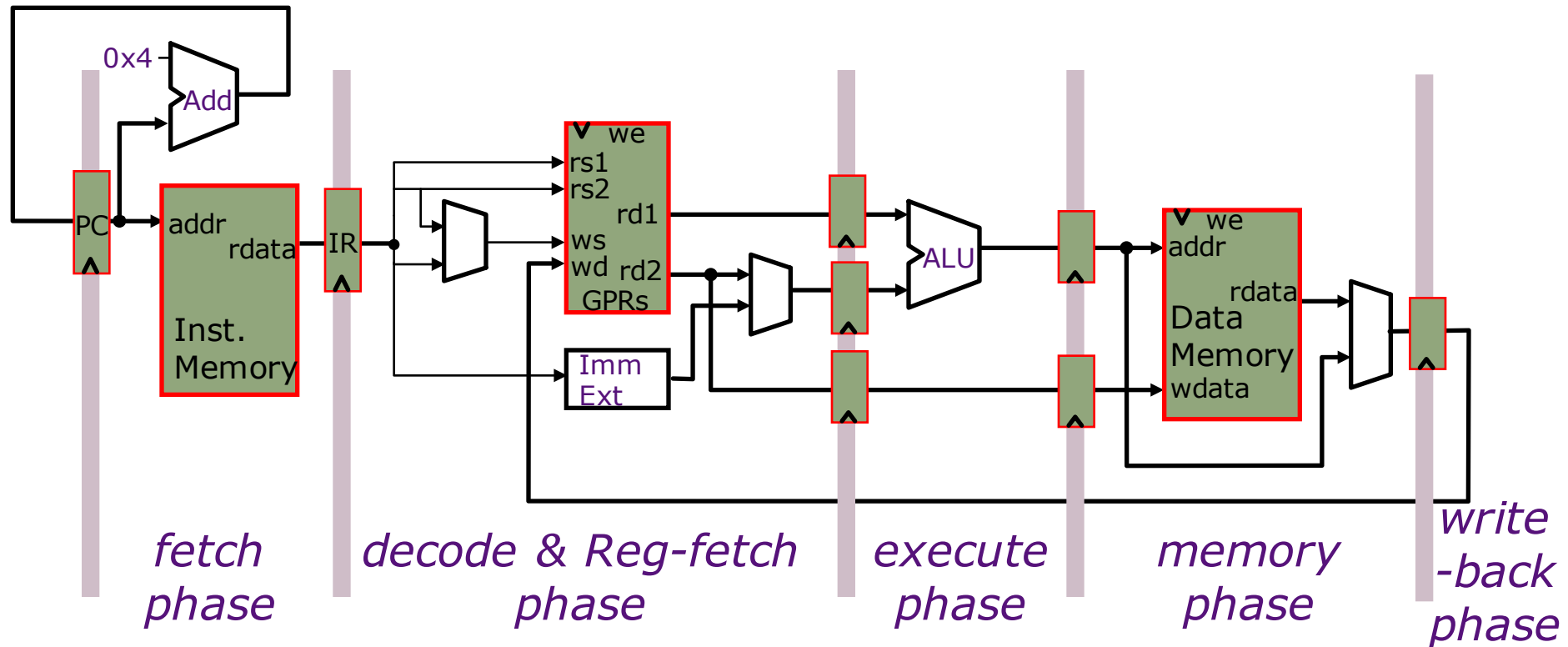
# Since Last Time...

## 1. Complex Pipelines

- Superscalar execution
- Out-of-order (OoO) processing
  - Scoreboarding
  - OoO: Issue, completion, retiring
  - Register renaming

## 2. Branch Prediction

# In-Order Pipeline



# In-Order Pipeline Limitations

Observation: True data dependency stalls dispatch of younger instructions into functional (execution) units.

```
MUL  R3 <- R1, R2
ADD  R3 <- R3, R1
ADD  R1 <- R6, R7
MUL  R5 <- R6, R8
ADD  R7 <- R3, R5
```

```
LD   R3 <-R1 (0)
ADD  R3 <- R3, R1
ADD  R1 <- R6, R7
MUL  R5 <- R6, R8
ADD  R7 <- R3, R5
```

# Let's take a step back: What limits performance?

## 1. Von Neumann Model

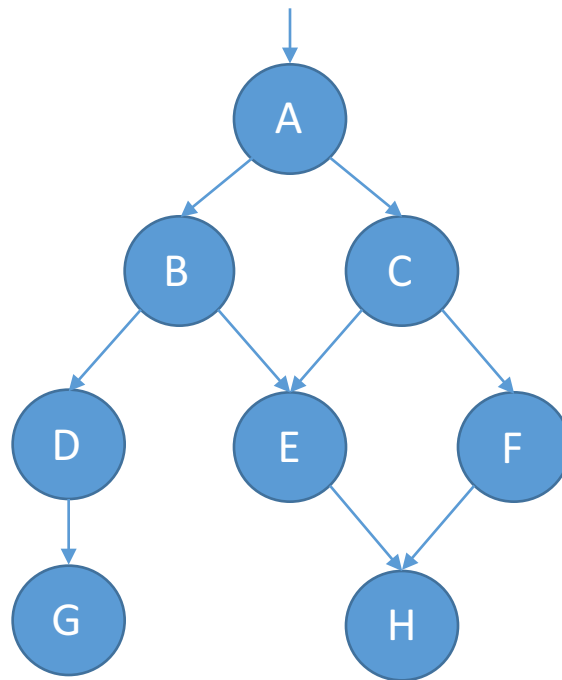
- Sequential stream of instructions

## 2. Implementation Issues

- Multi-cycle operations
- Variable latency operations

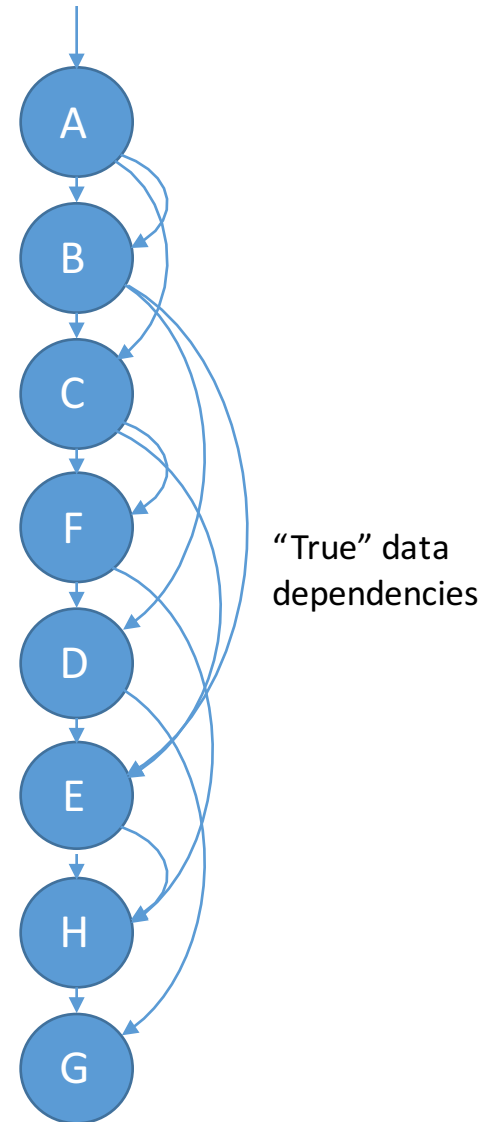
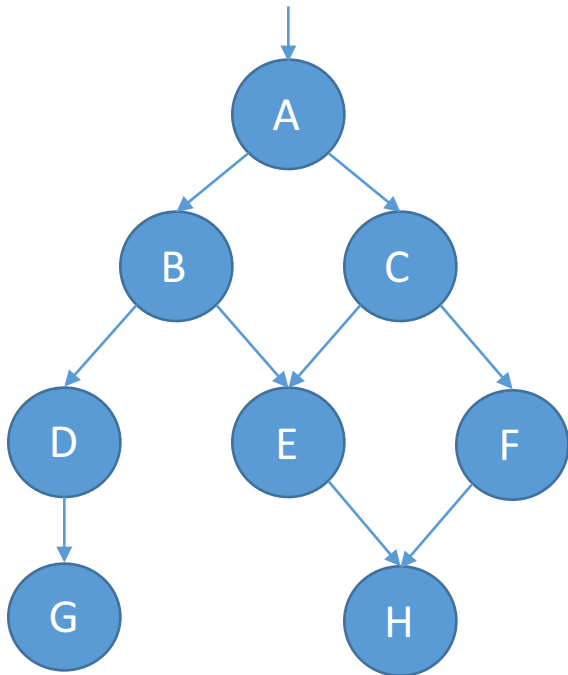
# Computation Structure

Every algorithm is conceptually a number of tasks with dependencies between them.



# Compilation

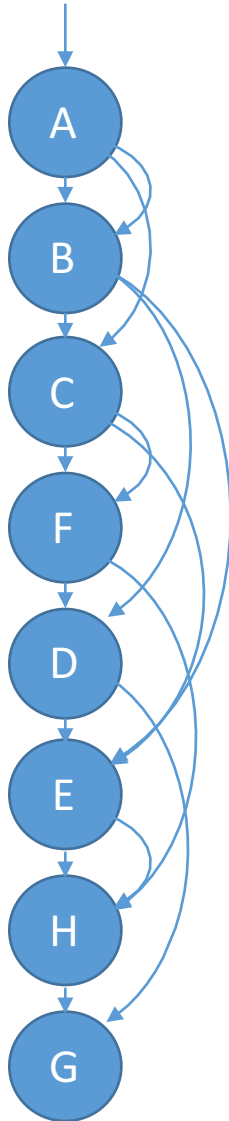
Compilation serializes this graph in some way



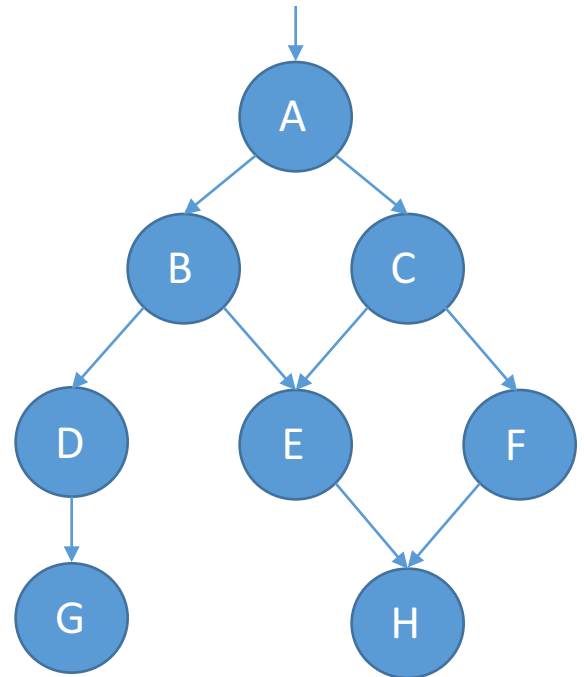
"True" data dependencies

In-order semantics—"false" dependencies

# Out-of-order Processing



Essentially, OOO tries to *dynamically* recover the true computation graph.





# How to do this correctly?

1. Must recognize dependencies between instructions

2. Must cause correct sequencing of the dependent instructions

3. Allow independent sequences of instructions to proceed concurrently



Correctness




Performance

# Dependencies

Data-dependence


$r_3 \leftarrow (r_1) \text{ op } (r_2)$   
 $r_5 \leftarrow (r_3) \text{ op } (r_4)$



Read-after-Write  
(RAW) hazard

Anti-dependence


$r_3 \leftarrow (r_1) \text{ op } (r_2)$   
 $r_1 \leftarrow (r_4) \text{ op } (r_5)$



Write-after-Read  
(WAR) hazard

Output-dependence

$r_3 \leftarrow (r_1) \text{ op } (r_2)$   
 $r_3 \leftarrow (r_6) \text{ op } (r_7)$



Write-after-Write  
(WAW) hazard

# OoO Issue / Dispatch

Stall until sure that issuing will cause no dependences

- Is the required functional unit available?
- Is the input data available?
- Is it safe to write the destination

Dependencies due to registers can be determined at decode stage.  
Data hazards due to memory operands can be determined only after computing effective address

# OoO Implementation

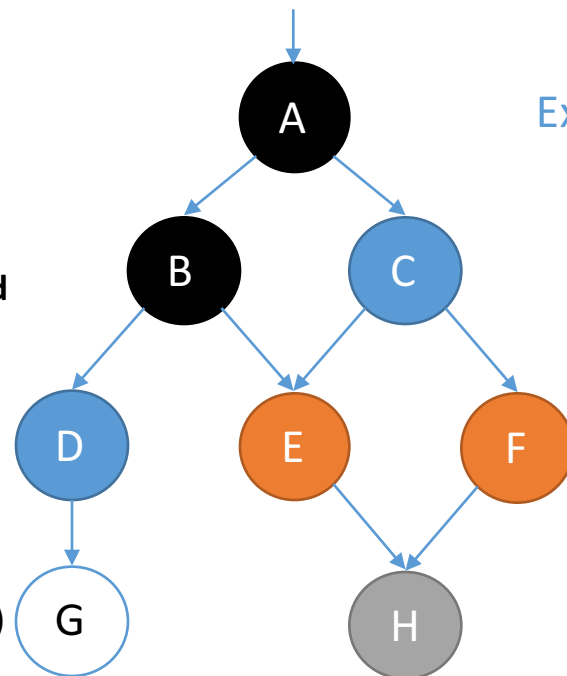
OoO implemented via a *re-order buffer* (ROB):

- ROB remembers original program order for in-order commit.
- ROB stores computation graph by its edges—the dependencies.

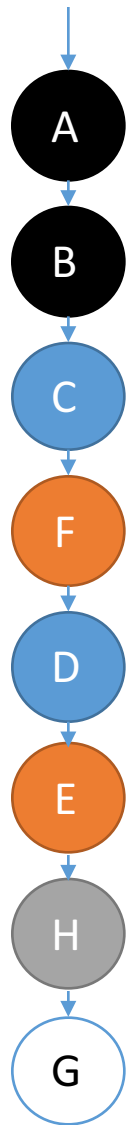
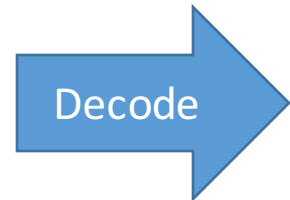
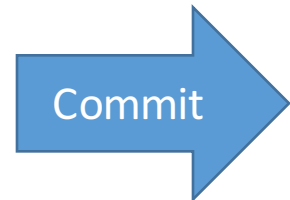
Task	State	Dependencies
A	Committed	
B	Committed	A
C	Completed	A
F	Ready	C
D	Completed	B
E	Ready	B, C
H	Waiting	E, F

Not ready  
Ready  
Complete  
Committed

(Not decoded)



Execute



# Is this sufficient?

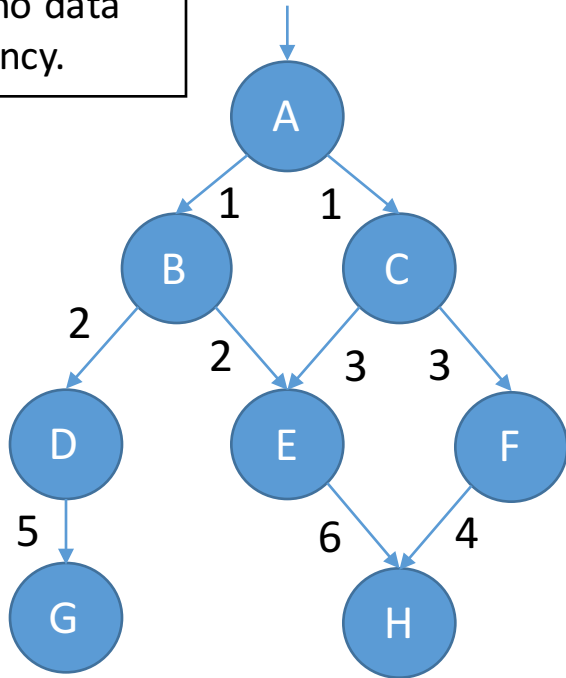
Number of registers limits maximum number of instructions in the pipeline.

WAW, WAR hazards are false dependencies introduced by limited number of architectural registers

# False dependencies

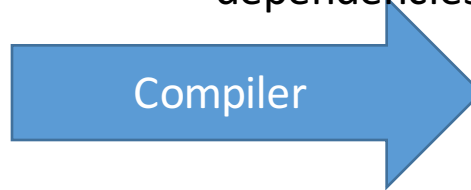
4 now overwrites 1 since both use R1, so we can't execute F before B even though there is no data dependency.

Register renaming helps remove false data dependencies.

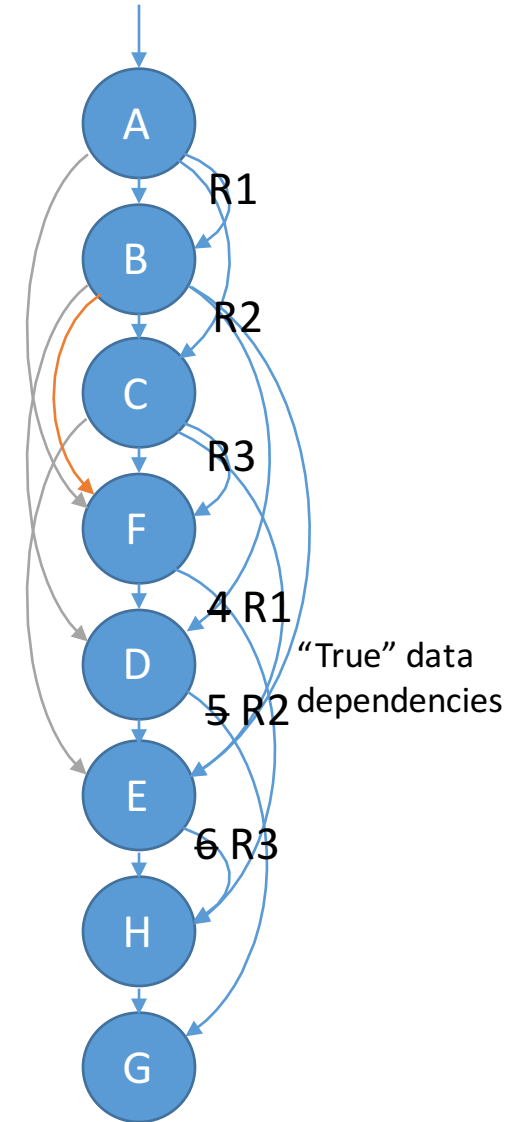


1. Give data dependencies names...

3. Register conflicts arise—  
"false"  
dependencies



2. Given three architectural registers: R1, R2, R3



In-order semantics—"false" dependencies

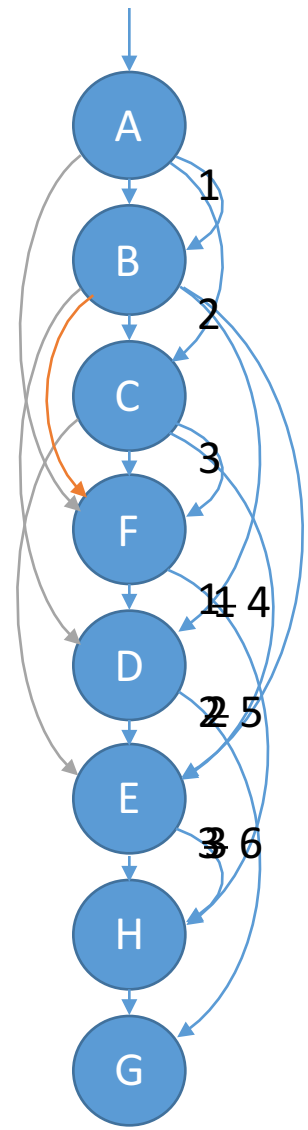
# Register Renaming

- *Register renaming* eliminates false dependencies by allocating a new register on every write.
- Requires many more “physical registers” than architectural registers and a layer of indirection.
  - Can think of architectural registers as “virtual registers” with the renaming table acting as a “register page table”.
- As before, the idea is to recover the *computation’s true structure* from the *over constrained* compiled code.

# Register Renaming

Hooray we recovered the original register names!  
 → No false dependencies!

Architectural Register	Physical Register
1	4
2	5
3	6



“True” data dependencies

In-order semantics—“false” dependencies



# OoO Implementation w/ renaming

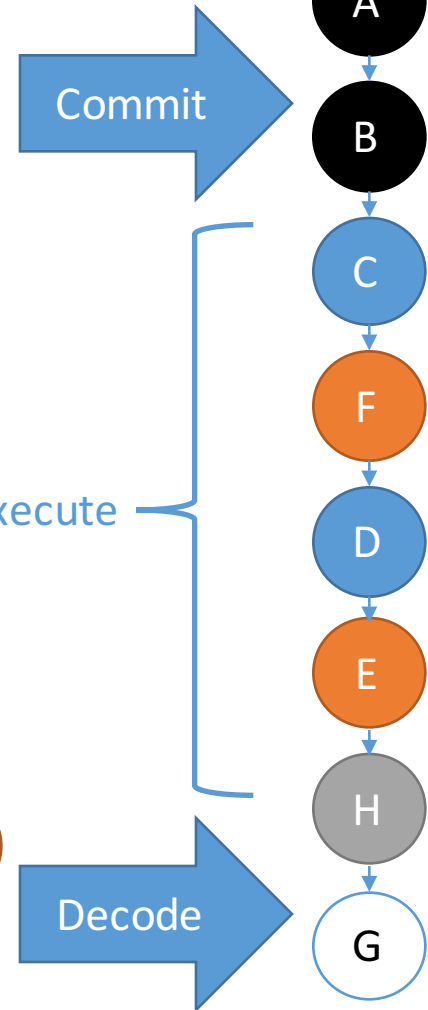
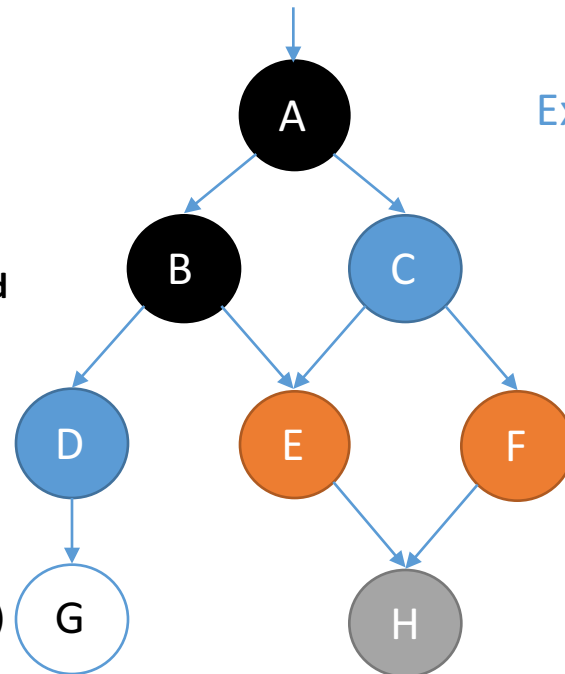
Architectural Register	Physical Register
R1	P1 P4
R2	P2 P5
R3	P3 P6

Express dependencies in terms of the *physical registers* that pass the data between instructions.

Task	State	Inputs	Output
A	Committed		P1
B	Committed	P1	P2
C	Completed	P1	P3
F	Ready	P3	P4
D	Completed	P2	P5
E	Ready	P2,P3	P6
H	Waiting	P4,P6	

Not ready  
**Ready**  
 Complete  
 Committed

(Not decoded)



# OoO: Summary

- OoO Processor: Restricted “data-flow” machine
  - Dynamically builds the data-flow graph
- The dynamically constructed data-flow graph is limited to the instruction window
- Tolerates long latency operations by executing independent instructions concurrently

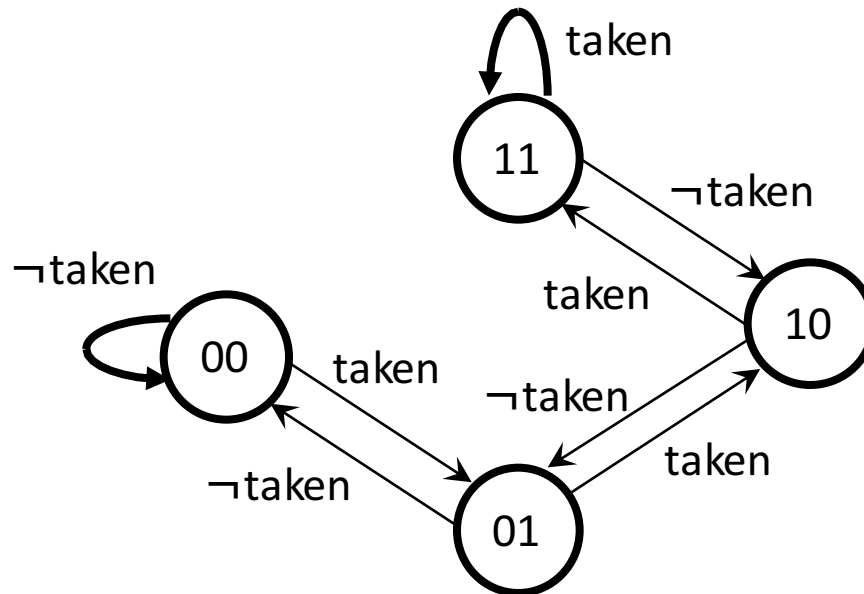
# Branch Prediction

Control Flow Dependences. How to handle them?

- Stall: Delay until we know the next PC
- Speculate: Guess next value
- Do something else: Multi-threading

# Branch Predictors

- 1-bit predictor
- 2-bit predictor

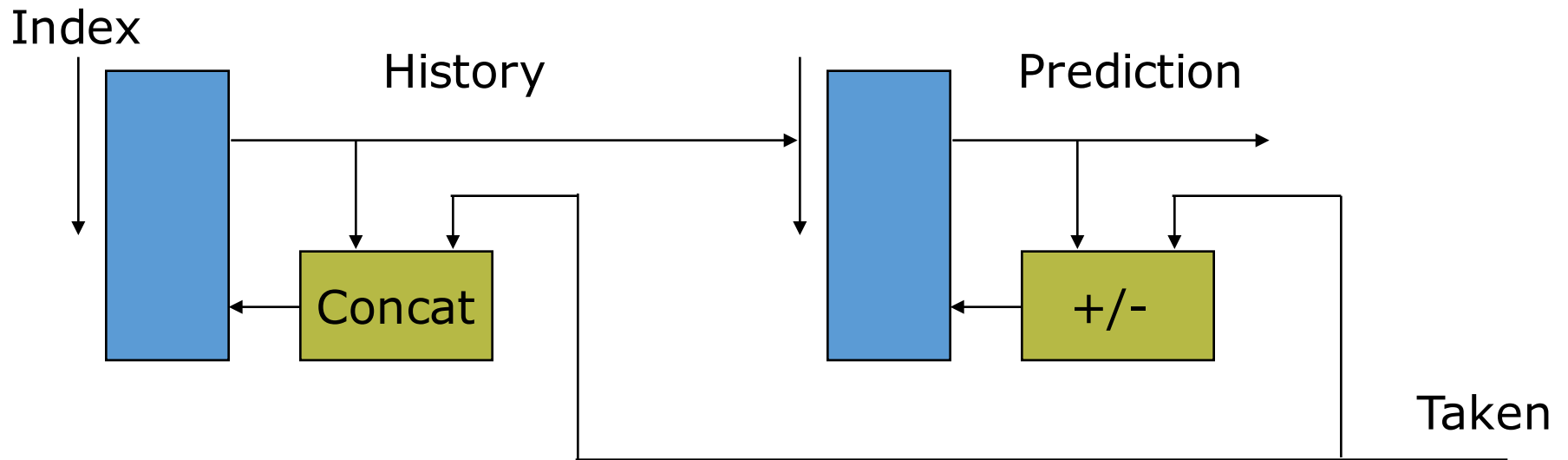


# Branch Predictors

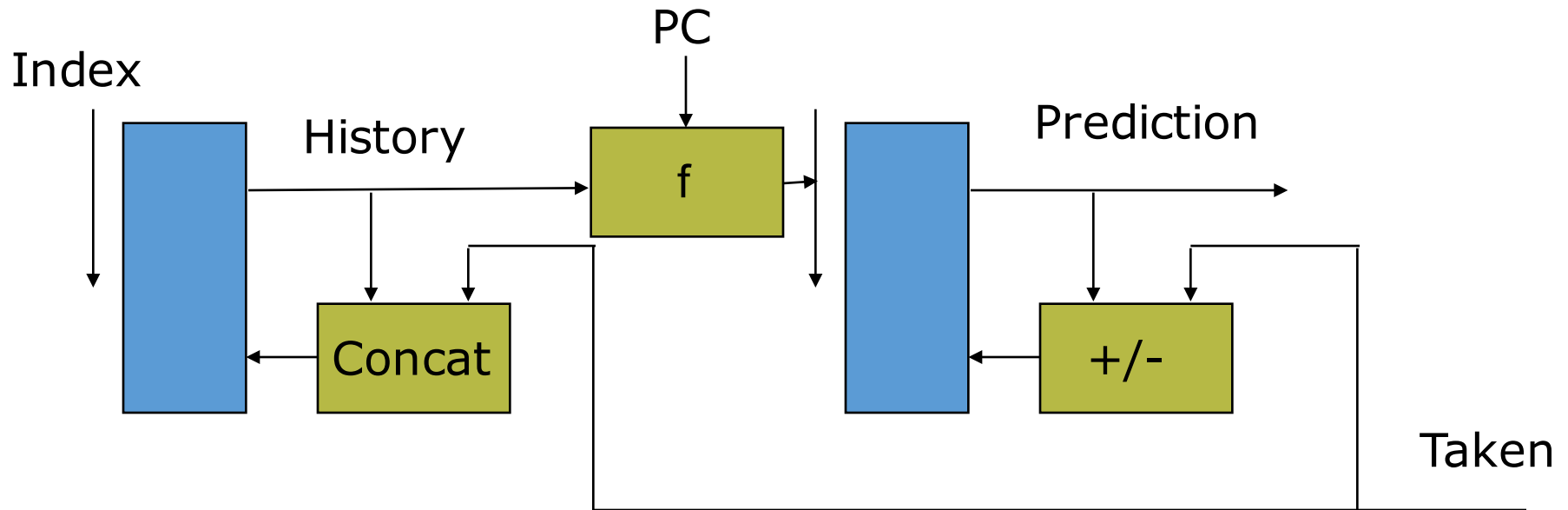
Two empirical observations

1. A branch's outcome can be correlated with other branches' outcomes
  - Global branch correlation
2. A branch's outcome can be correlated with past outcomes of the same branch
  - Local branch correlation

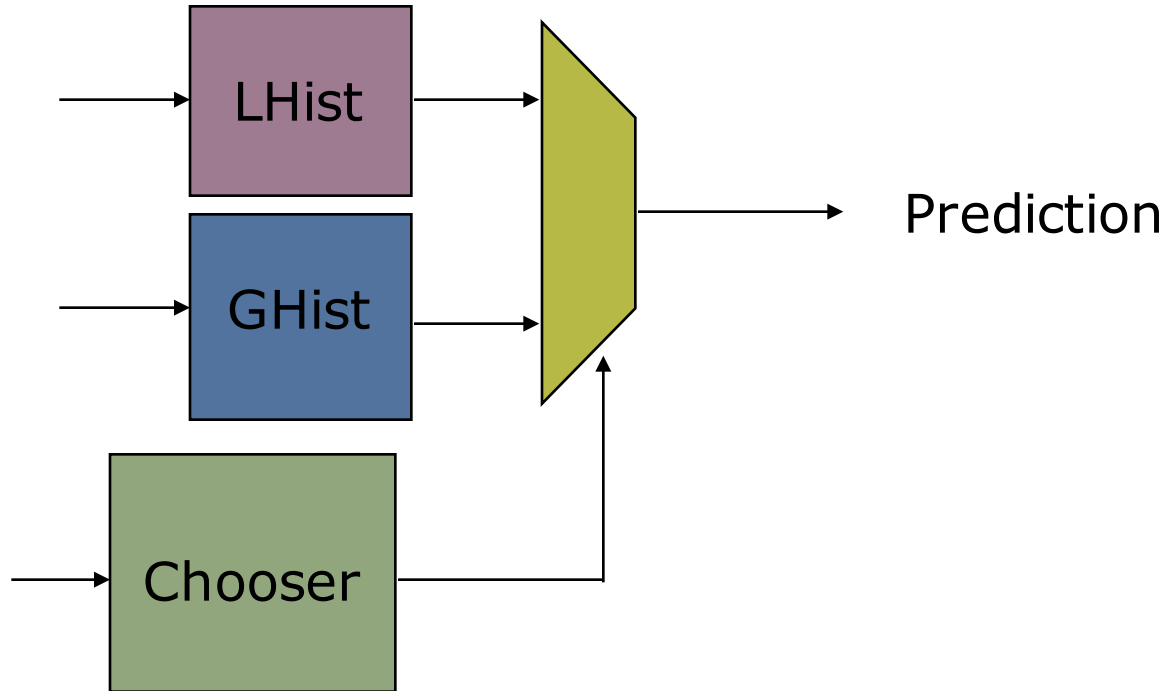
# History-based Prediction



# Two-level Predictor

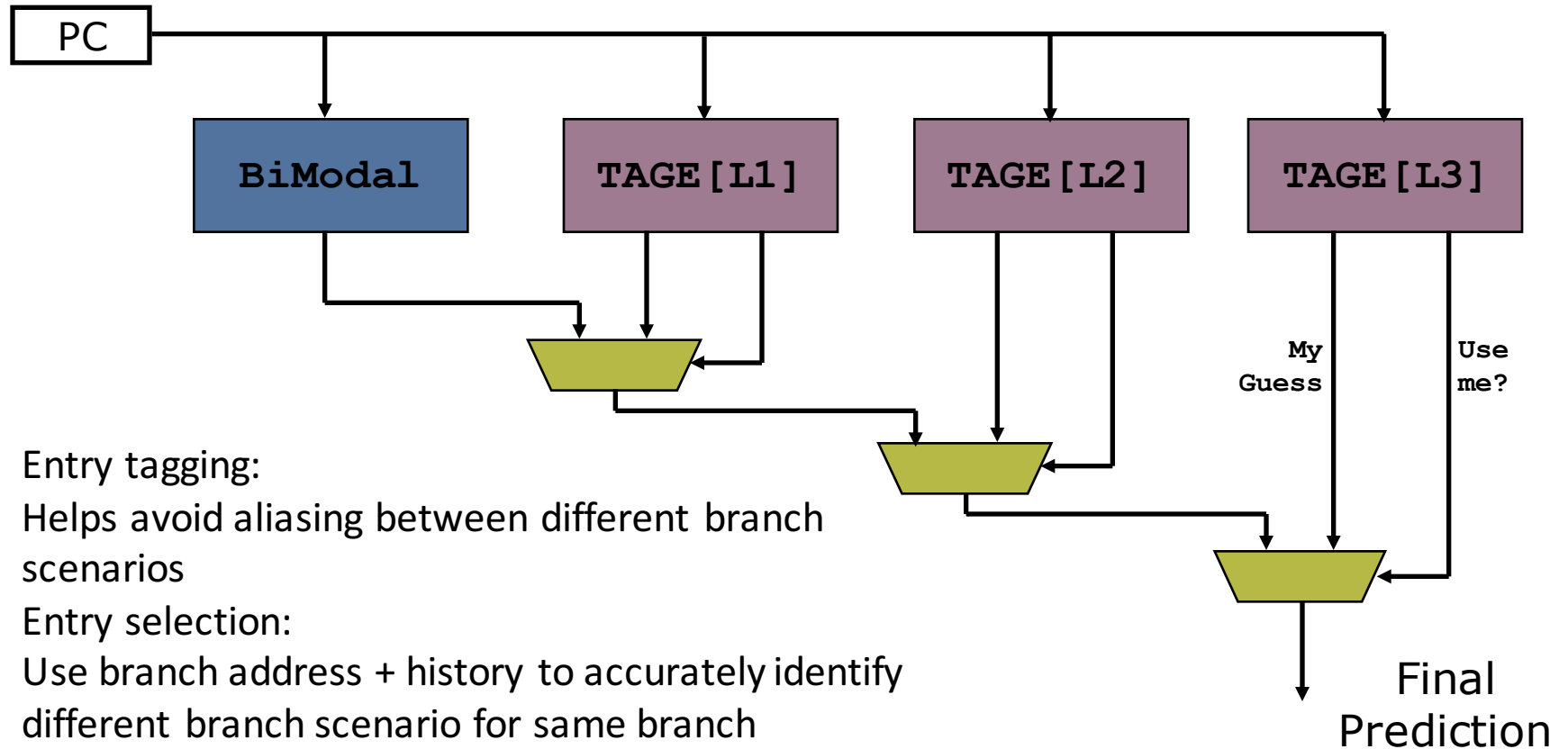


# Tournament Predictors





# TAGE Predictor



- Entry tagging:  
Helps avoid aliasing between different branch scenarios
- Entry selection:  
Use branch address + history to accurately identify different branch scenario for same branch
- Longer branch histories as required:  
Use long histories for branches that actually benefit