Modern Processor Architecture

Lecture Goals

- Learn about the key techniques that modern processors use to achieve high performance

Lecture Goals

- Learn about the key techniques that modern processors use to achieve high performance
- Emphasize the techniques that may help you in the design project (e.g., simple branch prediction)

Reminder: Processor Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}$$

CPI t_{CLK}

Reminder: Processor Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}$$

CPI t_{CLK}

- Pipelining lowers t_{CLK}. What about CPI?

Reminder: Processor Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}$$

CPI t_{CLK}

- Pipelining lowers t_{CLK}. What about CPI?
- $CPI = CPI_{ideal} + CPI_{hazard}$
  - CPI_{ideal}: cycles per instruction if no stalls

Reminder: Processor Performance

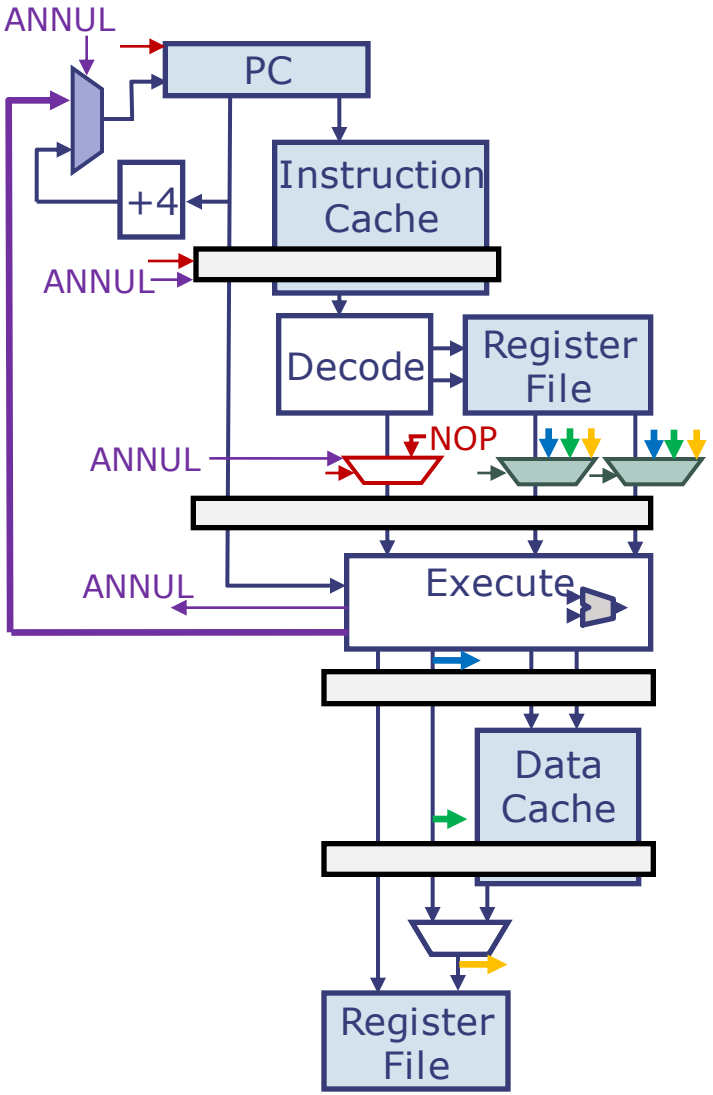
$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}$$

CPI t_{CLK}

- Pipelining lowers t_{CLK}. What about CPI?
- $CPI = CPI_{ideal} + CPI_{hazard}$
  - CPI_{ideal}: cycles per instruction if no stalls
- CPI_{hazard} contributors
  - Data hazards: long operations, cache misses
  - Control hazards: branches, jumps, exceptions

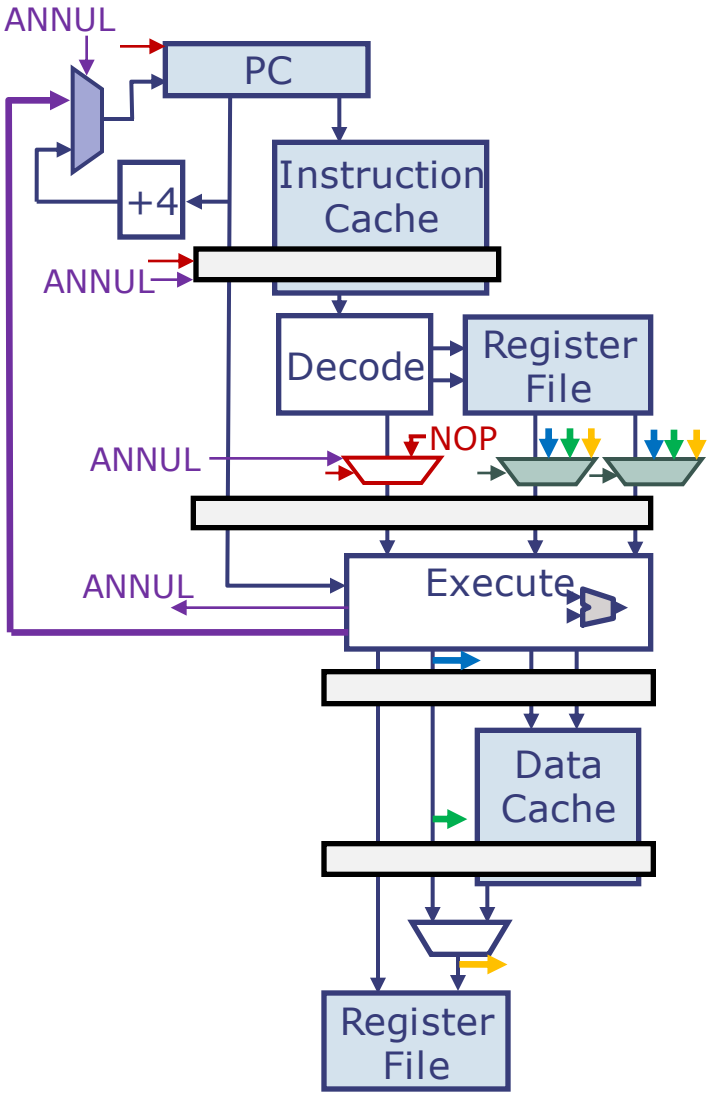
Standard 5-Stage Pipeline

- Assume full bypassing



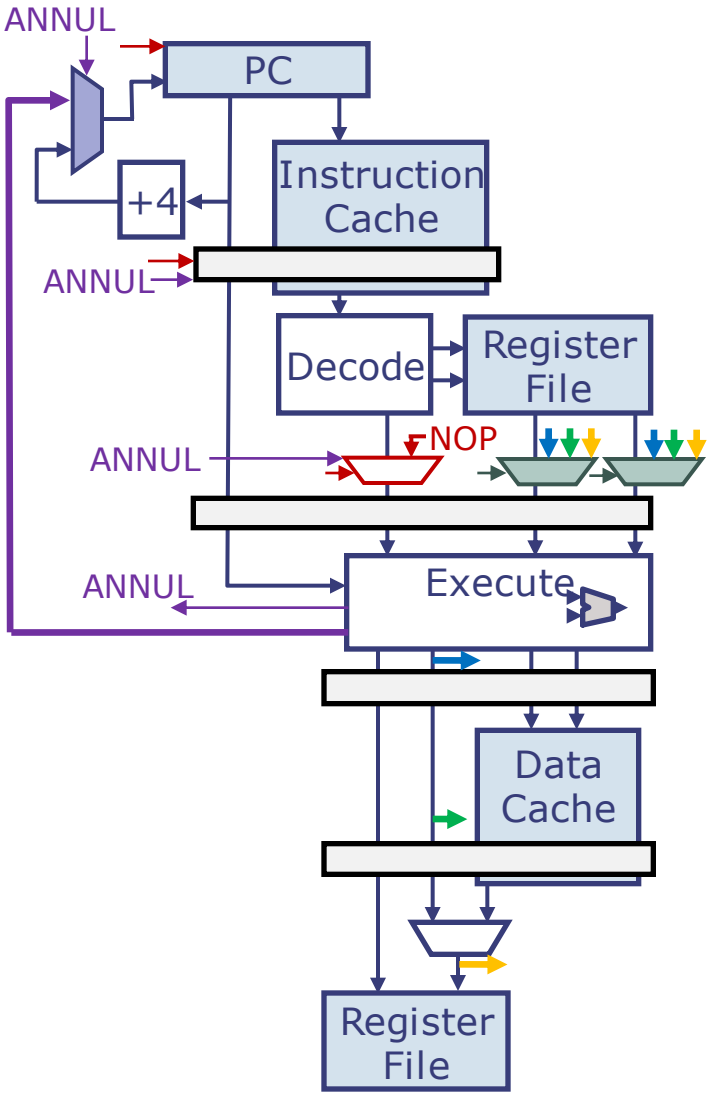
Standard 5-Stage Pipeline

- Assume full bypassing
- $CPI_{ideal}=1.0$



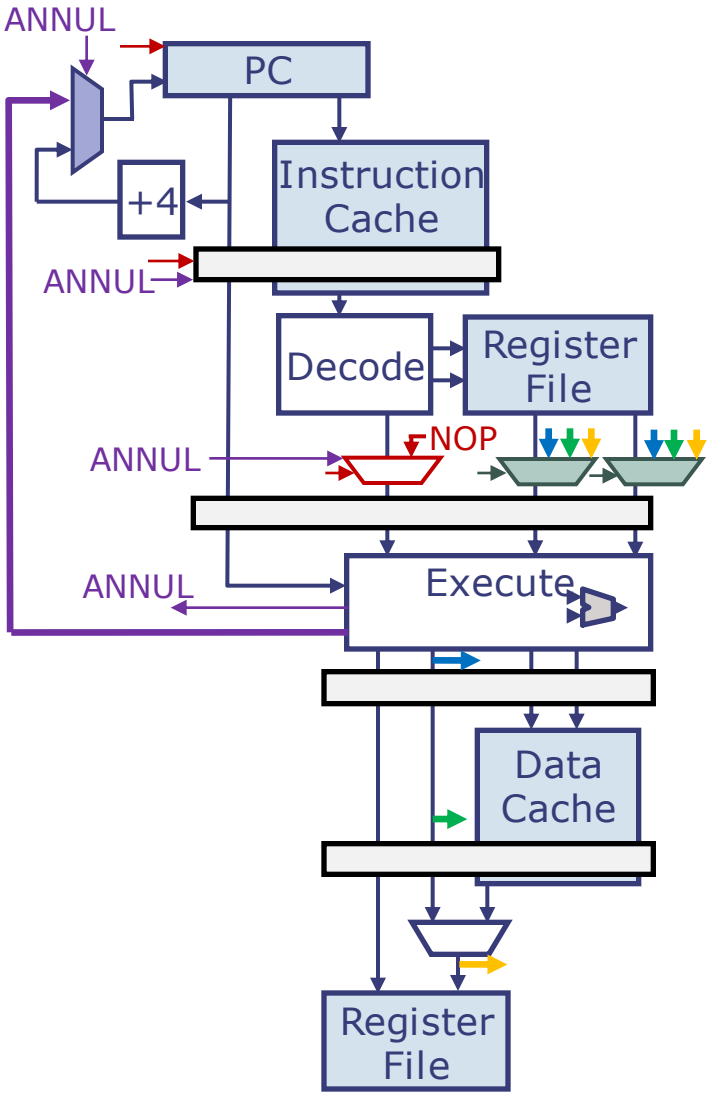
Standard 5-Stage Pipeline

- Assume full bypassing
- $CPI_{ideal}=1.0$
- CPI_{hazard} due to data hazards: *Up to how many cycles lost to each load-to-use hazard?* _____



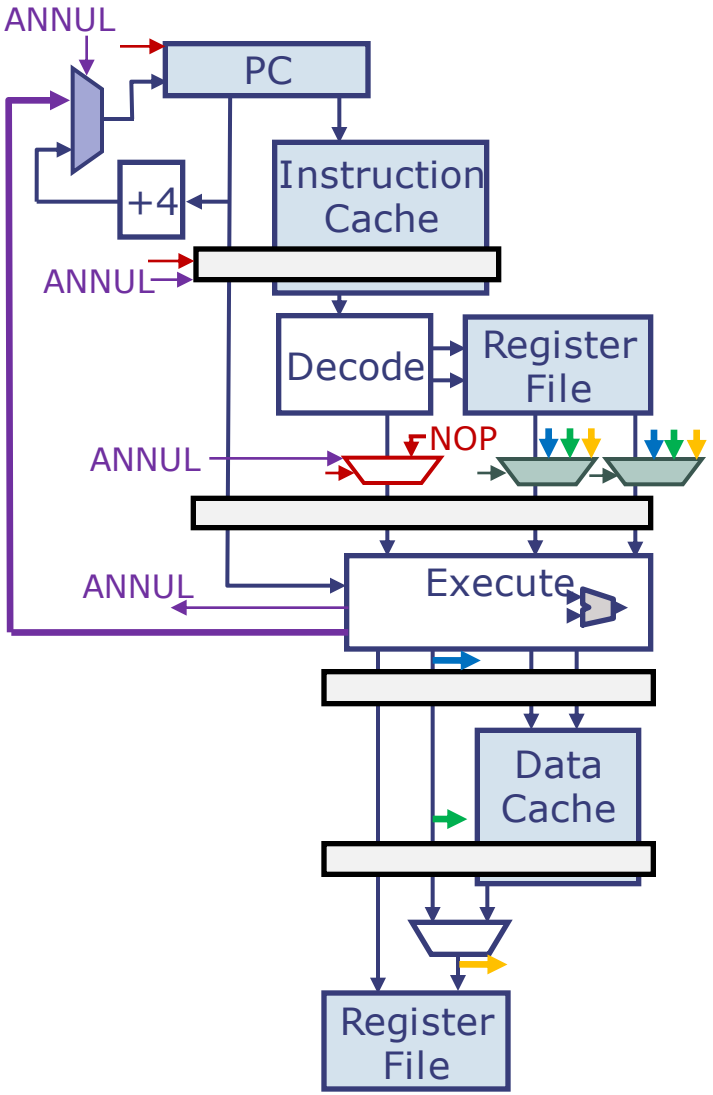
Standard 5-Stage Pipeline

- Assume full bypassing
- $CPI_{ideal}=1.0$
- CPI_{hazard} due to data hazards:
  Up to how many cycles lost to each load-to-use hazard? 2



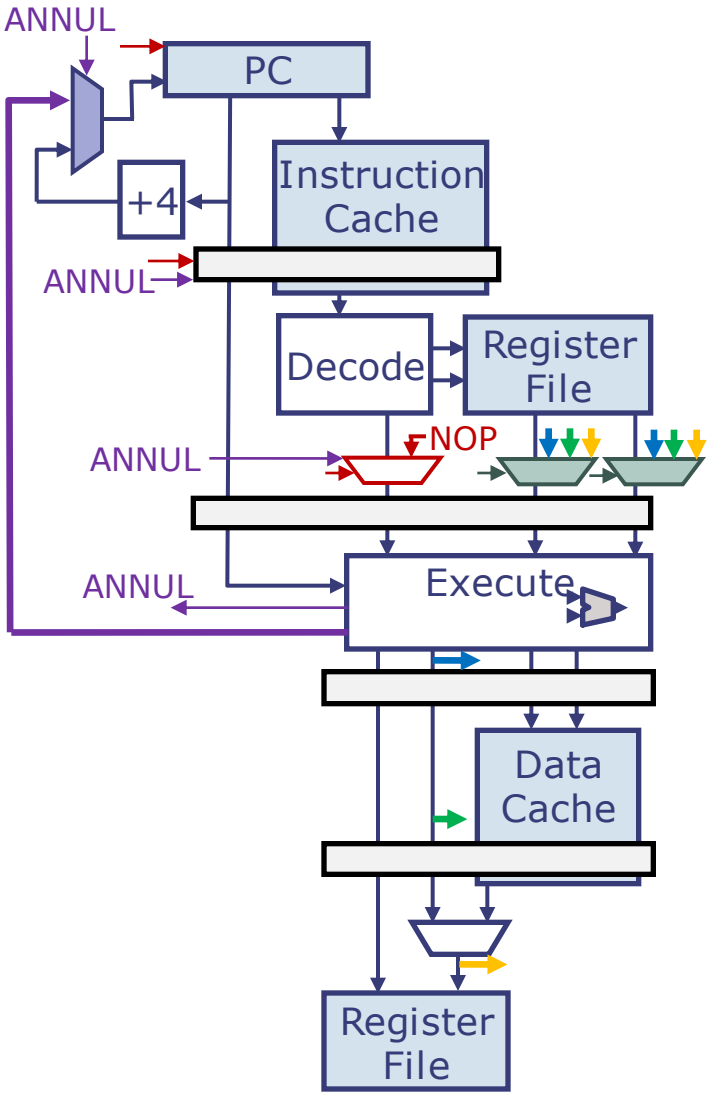
Standard 5-Stage Pipeline

- Assume full bypassing
- $CPI_{ideal}=1.0$
- CPI_{hazard} due to data hazards:
  Up to how many cycles lost to each load-to-use hazard? 2
- CPI_{hazard} due to control hazards:
  How many cycles lost to each jump and taken branch? _____



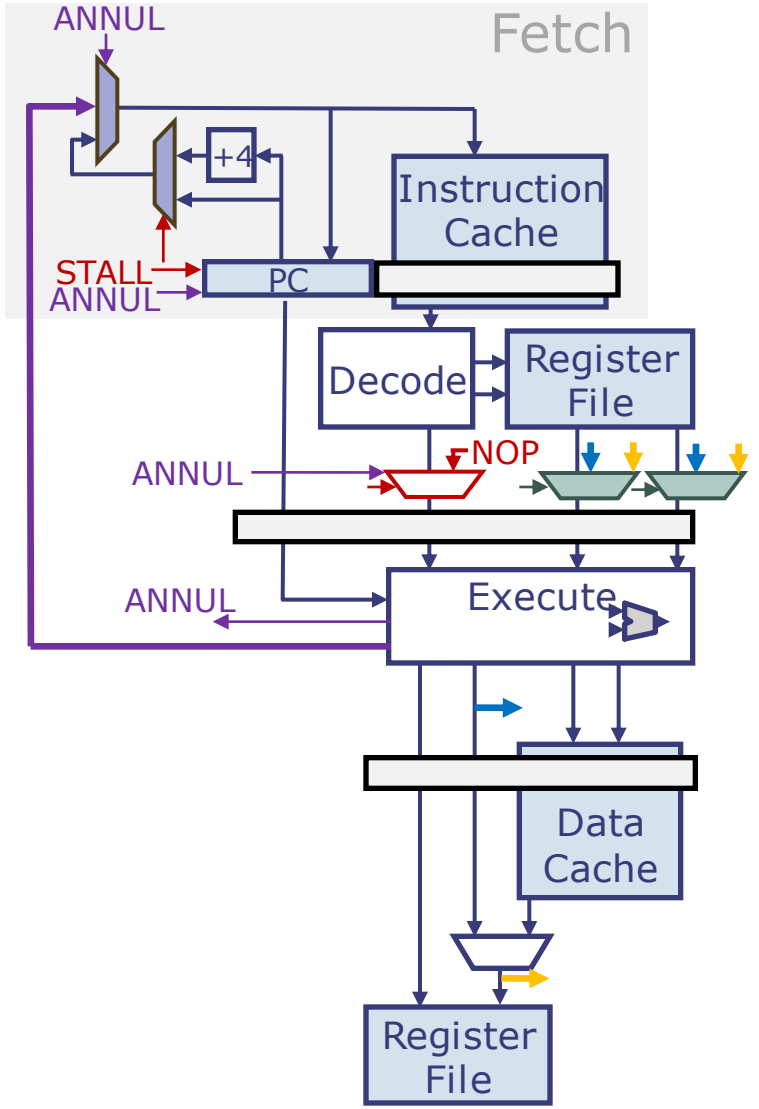
Standard 5-Stage Pipeline

- Assume full bypassing
- $CPI_{ideal}=1.0$
- CPI_{hazard} due to data hazards:
  Up to how many cycles lost to each load-to-use hazard? 2
- CPI_{hazard} due to control hazards:
  How many cycles lost to each jump and taken branch? 2



Design Project Pipeline (Part 2)

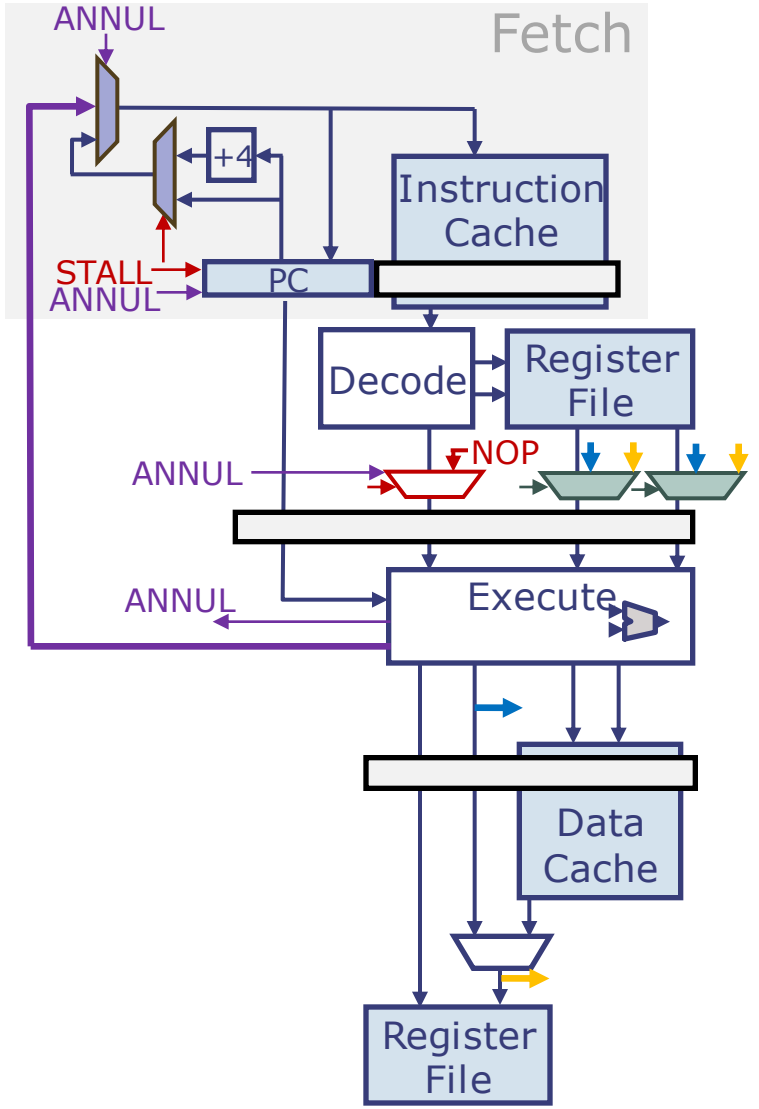
- 4 stages: IF, DEC, EXE, WB
  - No MEM stage
- IF uses *PC bypassing*: On annulment, IF starts fetching at the jump/branch target on the same cycle



Design Project Pipeline (Part 2)

- 4 stages: IF, DEC, EXE, WB
  - No MEM stage
- IF uses *PC bypassing*: On annulment, IF starts fetching at the jump/branch target on the same cycle

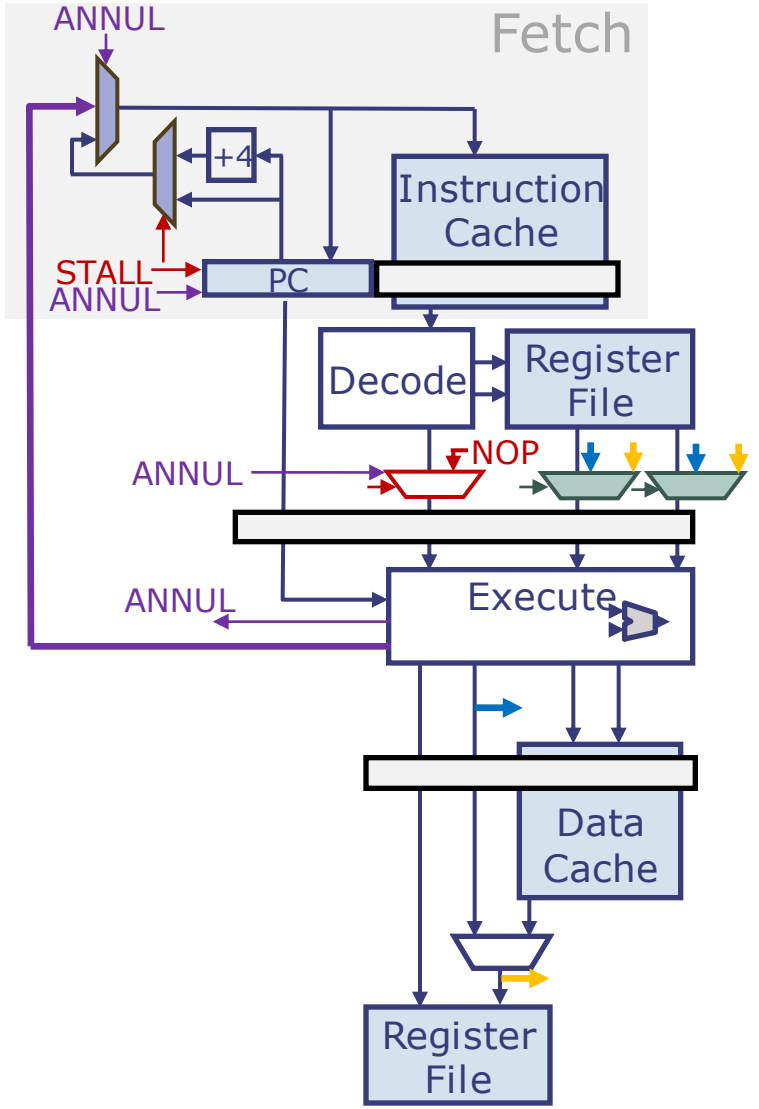
- CPI_{hazard} due to data hazards: *Up to how many cycles lost to each load-to-use hazard?* _____



Design Project Pipeline (Part 2)

- 4 stages: IF, DEC, EXE, WB
  - No MEM stage
- IF uses *PC bypassing*: On annulment, IF starts fetching at the jump/branch target on the same cycle

- CPI_{hazard} due to data hazards: *Up to how many cycles lost to each load-to-use hazard?* 1

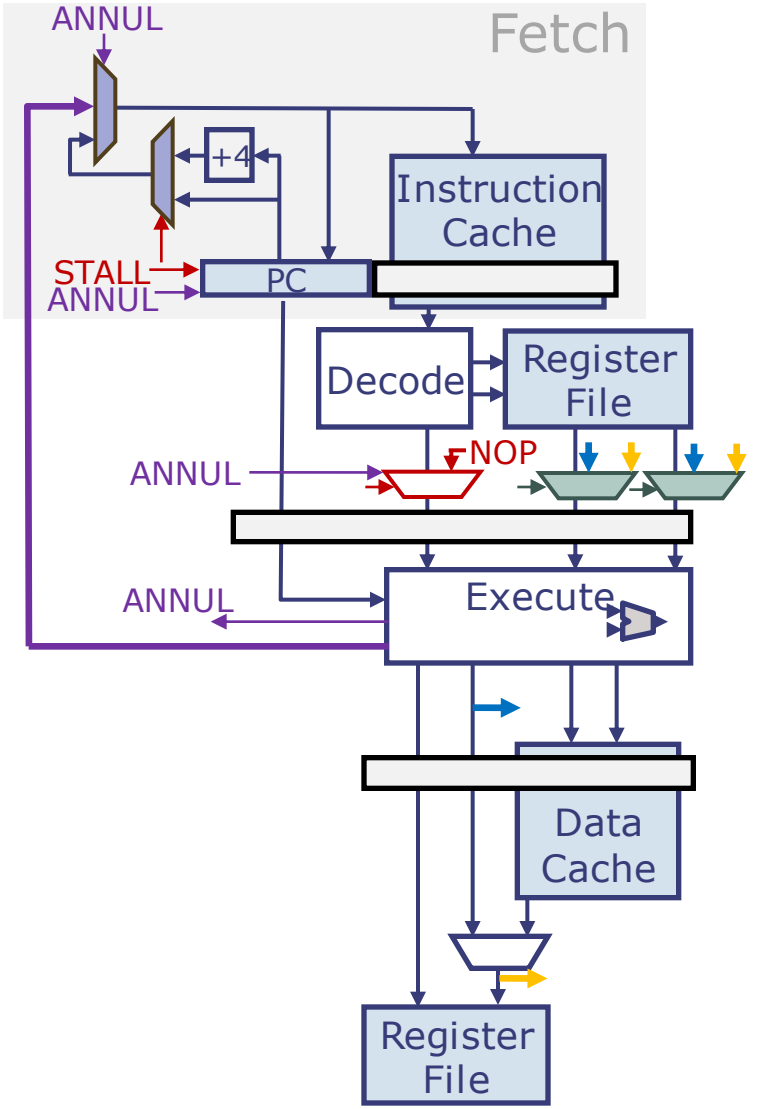


Design Project Pipeline (Part 2)

- 4 stages: IF, DEC, EXE, WB
  - No MEM stage
- IF uses *PC bypassing*: On annulment, IF starts fetching at the jump/branch target on the same cycle

- CPI_{hazard} due to data hazards: *Up to how many cycles lost to each load-to-use hazard?* 1

- CPI_{hazard} due to control hazards: *How many cycles lost to each jump and taken branch?* _____

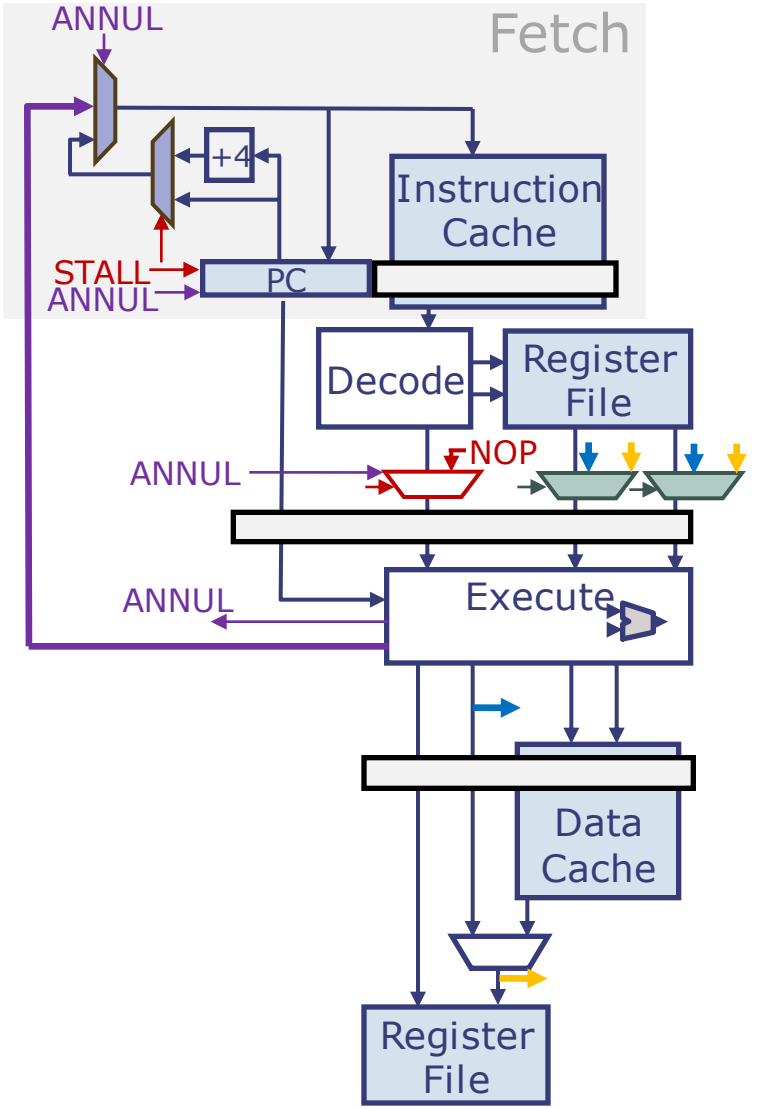


Design Project Pipeline (Part 2)

- 4 stages: IF, DEC, EXE, WB
  - No MEM stage
- IF uses *PC bypassing*: On annulment, IF starts fetching at the jump/branch target on the same cycle

- CPI_{hazard} due to data hazards: *Up to how many cycles lost to each load-to-use hazard?* 1

- CPI_{hazard} due to control hazards: *How many cycles lost to each jump and taken branch?* 1



Improving Processor Performance

- Increase clock frequency: deeper pipelines
  - Overlap more instructions

Improving Processor Performance

- Increase clock frequency: deeper pipelines
  - Overlap more instructions
- Reduce CPI_{ideal}: wider pipelines
  - Each pipeline stage processes multiple instructions

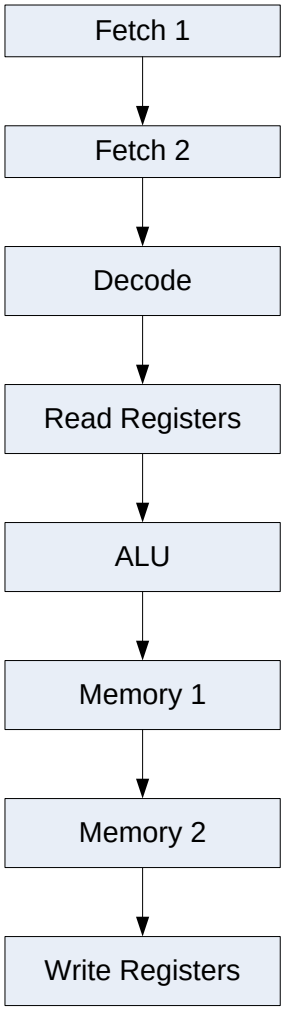
Improving Processor Performance

- Increase clock frequency: deeper pipelines
  - Overlap more instructions
- Reduce CPI_{ideal}: wider pipelines
  - Each pipeline stage processes multiple instructions
- Reduce impact of data hazards: out-of-order execution
  - Execute each instruction as soon as its source operands are available

Improving Processor Performance

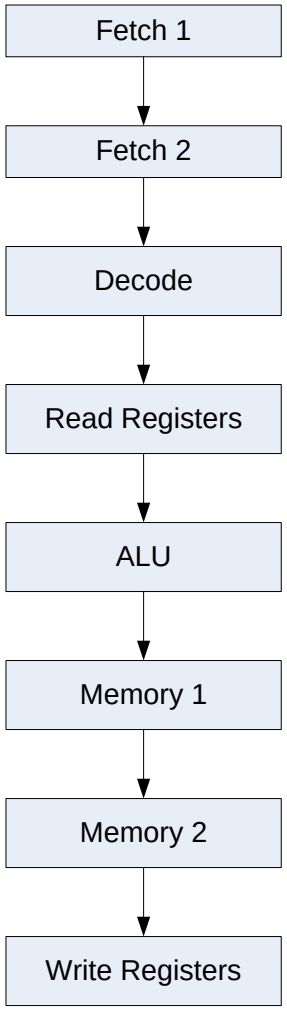
- Increase clock frequency: deeper pipelines
  - Overlap more instructions
- Reduce CPI_{ideal}: wider pipelines
  - Each pipeline stage processes multiple instructions
- Reduce impact of data hazards: out-of-order execution
  - Execute each instruction as soon as its source operands are available
- Reduce impact of control hazards: branch prediction
  - Predict both direction and target of branches and jumps

Deeper Pipelines



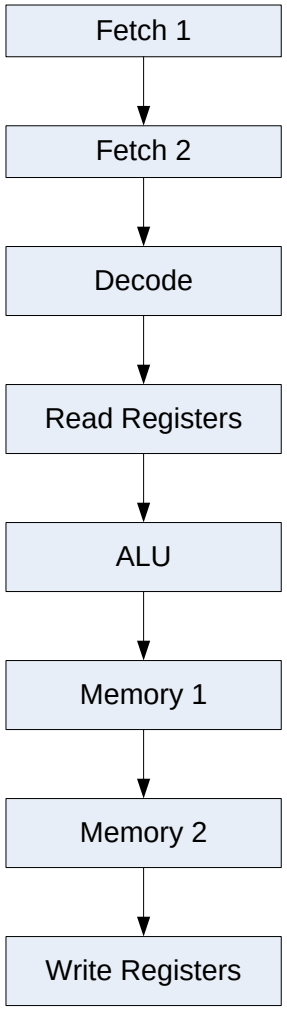
- Break up datapath into N pipeline stages
  - Ideal t_{CLK} = 1/N compared to non-pipelined
  - So let's use a large N!

Deeper Pipelines



- Break up datapath into N pipeline stages
  - Ideal t_{CLK} = 1/N compared to non-pipelined
  - So let's use a large N!
- Advantage: Higher clock frequency
  - The workhorse behind multi-GHz processors
  - Intel Skylake, AMD Zen2: 19 stages, 4-5 GHz

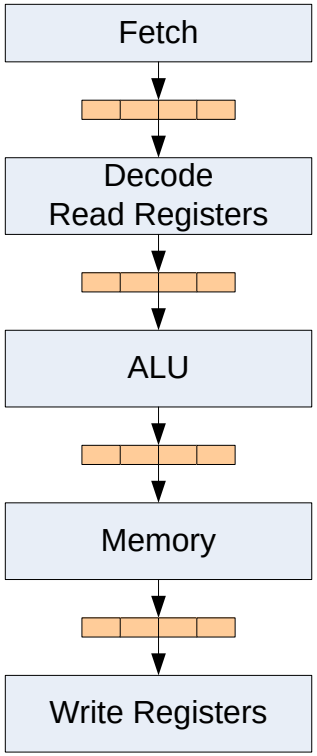
Deeper Pipelines



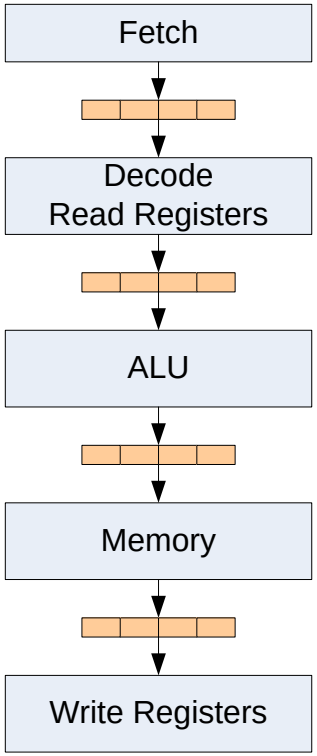
- Break up datapath into N pipeline stages
  - Ideal t_{CLK} = 1/N compared to non-pipelined
  - So let's use a large N!
- Advantage: Higher clock frequency
  - The workhorse behind multi-GHz processors
  - Intel Skylake, AMD Zen2: 19 stages, 4-5 GHz
- Disadvantages
  - More overlapping ⇒ more dependencies
    - CPI_{hazard} grows due to data and control hazards
  - Pipeline registers add area & power

Wider (aka Superscalar) Pipelines

- Each stage operates on up to W instructions each clock cycle

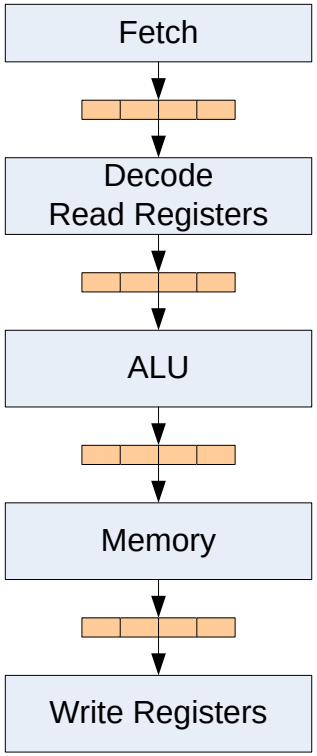


Wider (aka Superscalar) Pipelines



- Each stage operates on up to W instructions each clock cycle
- Advantage: Lower CPI_{ideal} (1/W)
  - Skylake & Zen2: 6-wide, Apple M1: 8-wide

Wider (aka Superscalar) Pipelines



- Each stage operates on up to W instructions each clock cycle
- Advantage: Lower CPI_{ideal} (1/W)
  - Skylake & Zen2: 6-wide, Apple M1: 8-wide
- Disadvantages
  - Parallel execution ⇒ more dependencies
    - CPI_{hazard} grows due to data and control hazards
  - Much higher cost & complexity
    - More ALUs, register file ports, …
    - Many bypass & stall cases to check

Resolving Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages
- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated
- Strategy 3: Speculate
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value

Resolving Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages
- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated
- Strategy 3: Speculate
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value
- Strategy 4: Find something else to do

Resolving Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages
- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated
- Strategy 3: Speculate
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value
- Strategy 4: Find something else to do

Out-of-Order Execution

- Consider the expression $D = 3(a - b) + 7ac$

Out-of-Order Execution

- Consider the expression $D = 3(a - b) + 7ac$

Sequential code

ld a
ld b
sub a-b
mul 3(a-b)
ld c
mul ac
mul 7ac
add 3(a-b)+7ac
st d

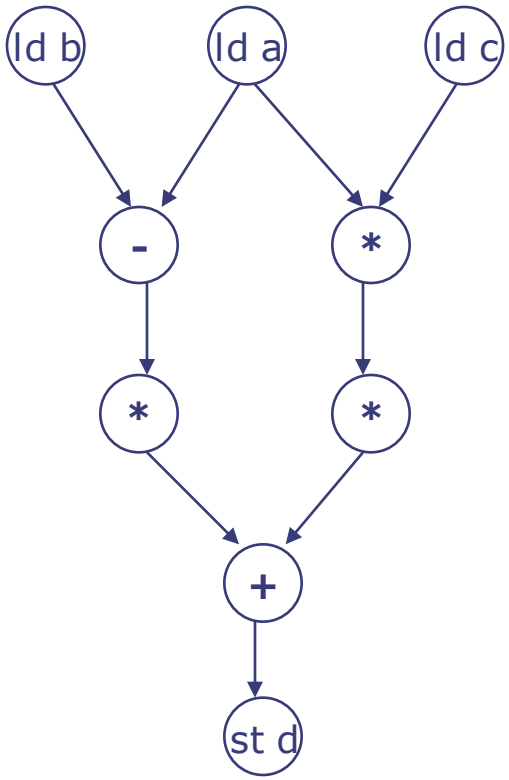
Out-of-Order Execution

- Consider the expression $D = 3(a - b) + 7ac$

Sequential code

ld a
ld b
sub a-b
mul 3(a-b)
ld c
mul ac
mul 7ac
add 3(a-b)+7ac
st d

Dataflow graph



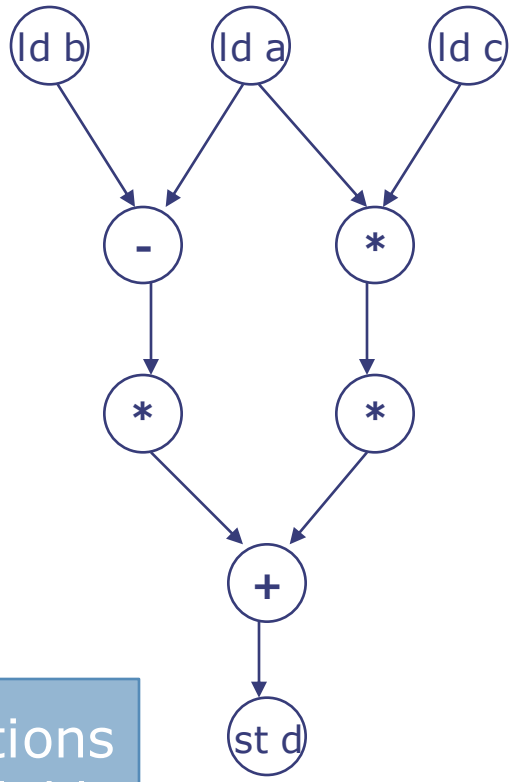
Out-of-Order Execution

- Consider the expression $D = 3(a - b) + 7ac$

Sequential code

```
ld a
ld b
sub a-b
mul 3(a-b)
ld c
mul ac
mul 7ac
add 3(a-b)+7ac
st d
```

Dataflow graph



Out-of-order execution runs instructions as soon as their inputs become available

Out-of-Order Execution Example

- If `ld b` takes a few cycles (e.g., cache miss), can execute instructions that do not depend on b

Sequential code

ld a
ld b
sub a-b
mul 3(a-b)
ld c
mul ac
mul 7ac
add 3(a-b)+7ac
st d

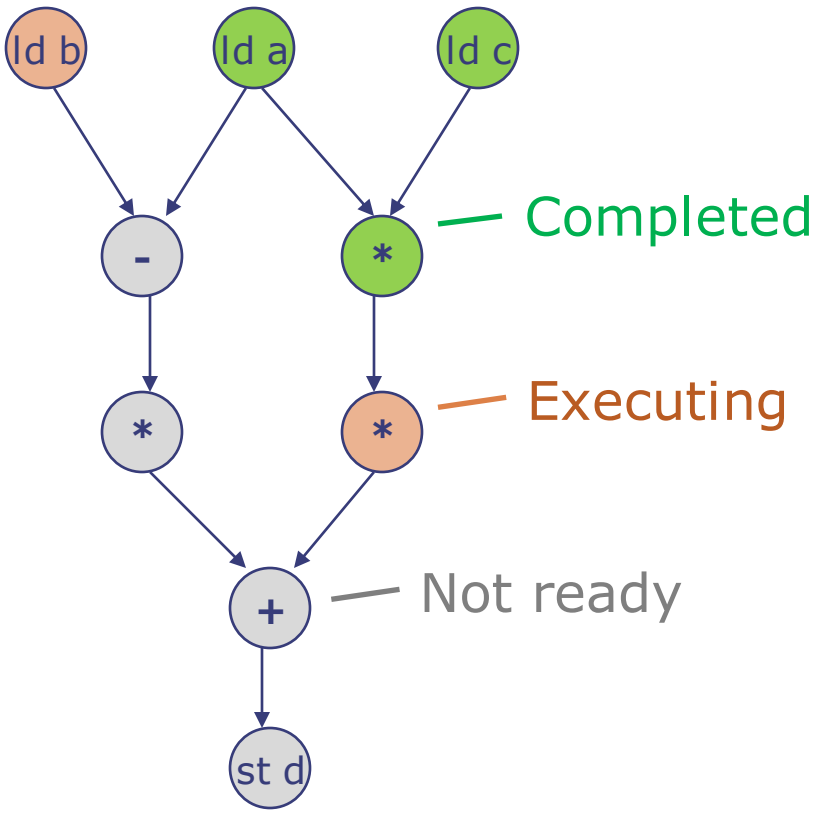
Out-of-Order Execution Example

- If `ld b` takes a few cycles (e.g., cache miss), can execute instructions that do not depend on b

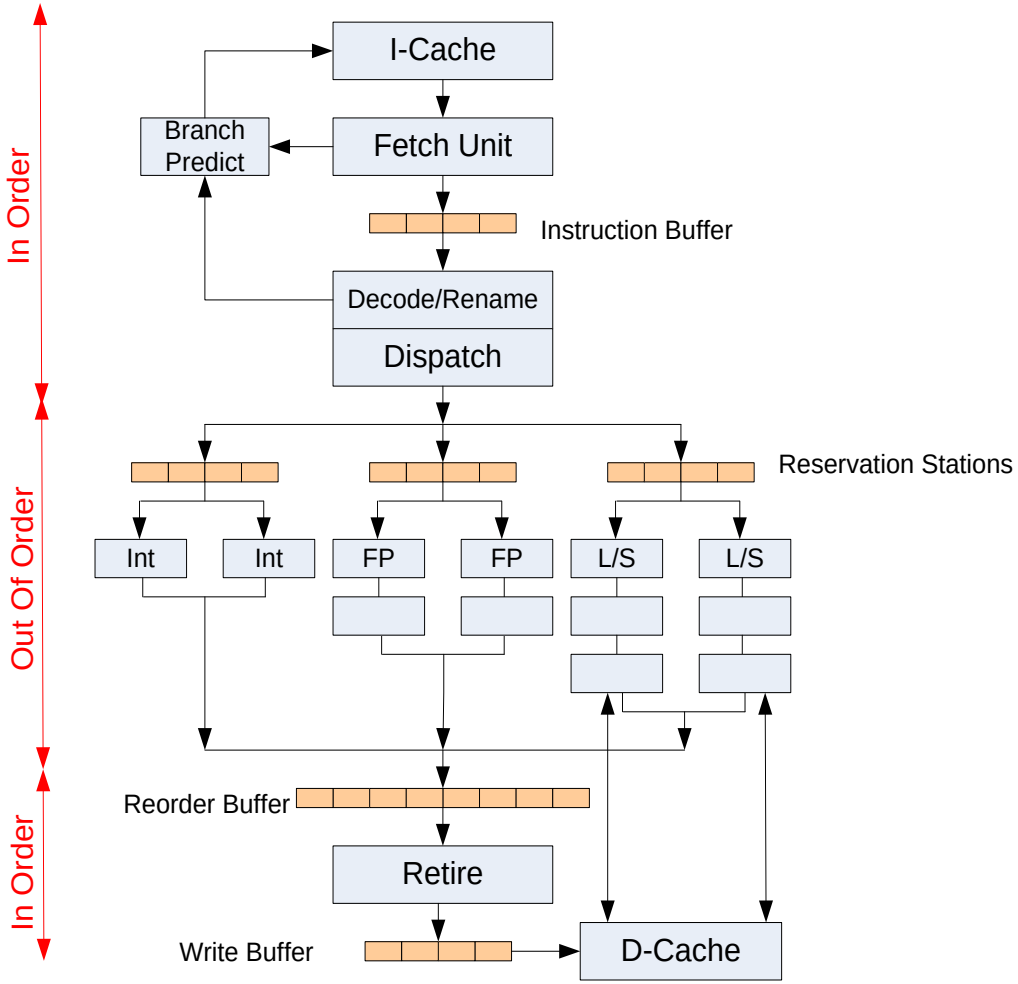
Sequential code

```
    ld a
➡   ld b
    sub a-b
    mul 3(a-b)
    ld c
    mul ac
    mul 7ac
    add 3(a-b)+7ac
    st d
```

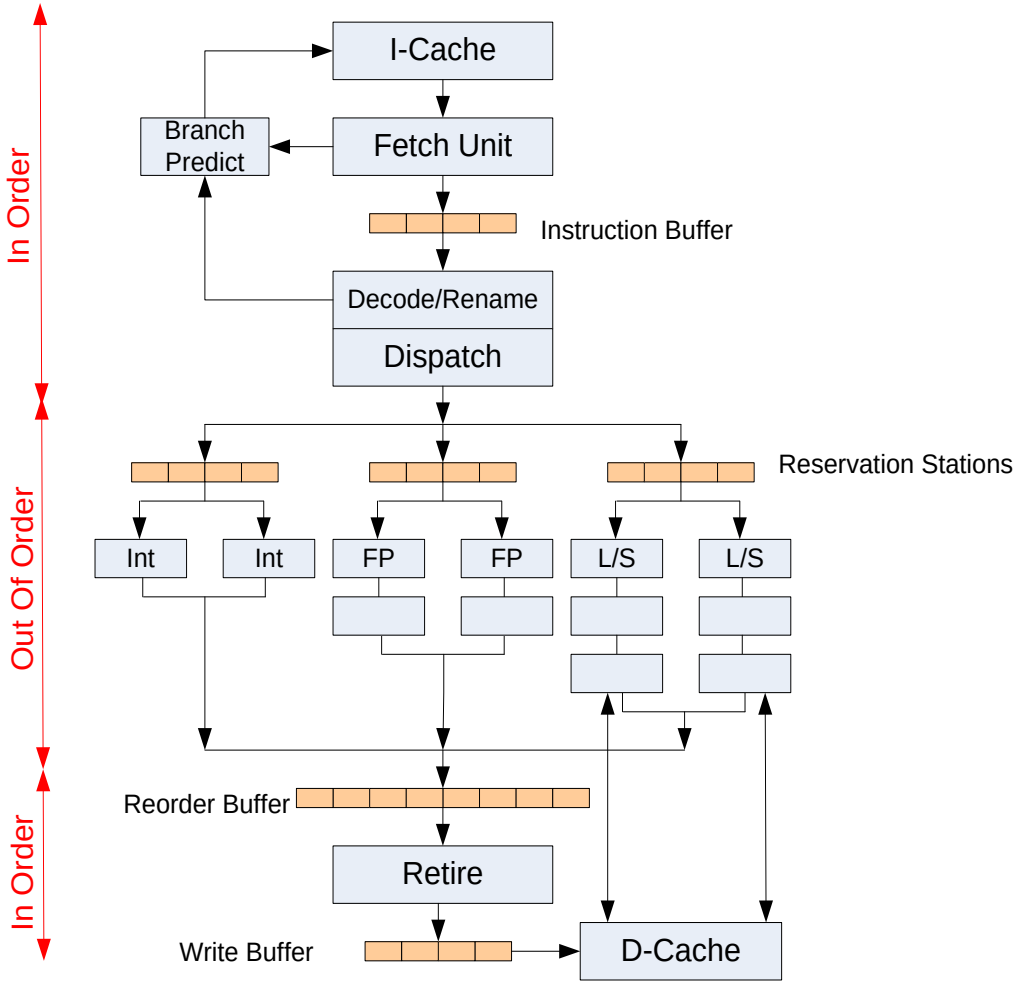
Dataflow graph



A Modern Out-of-Order Superscalar Processor

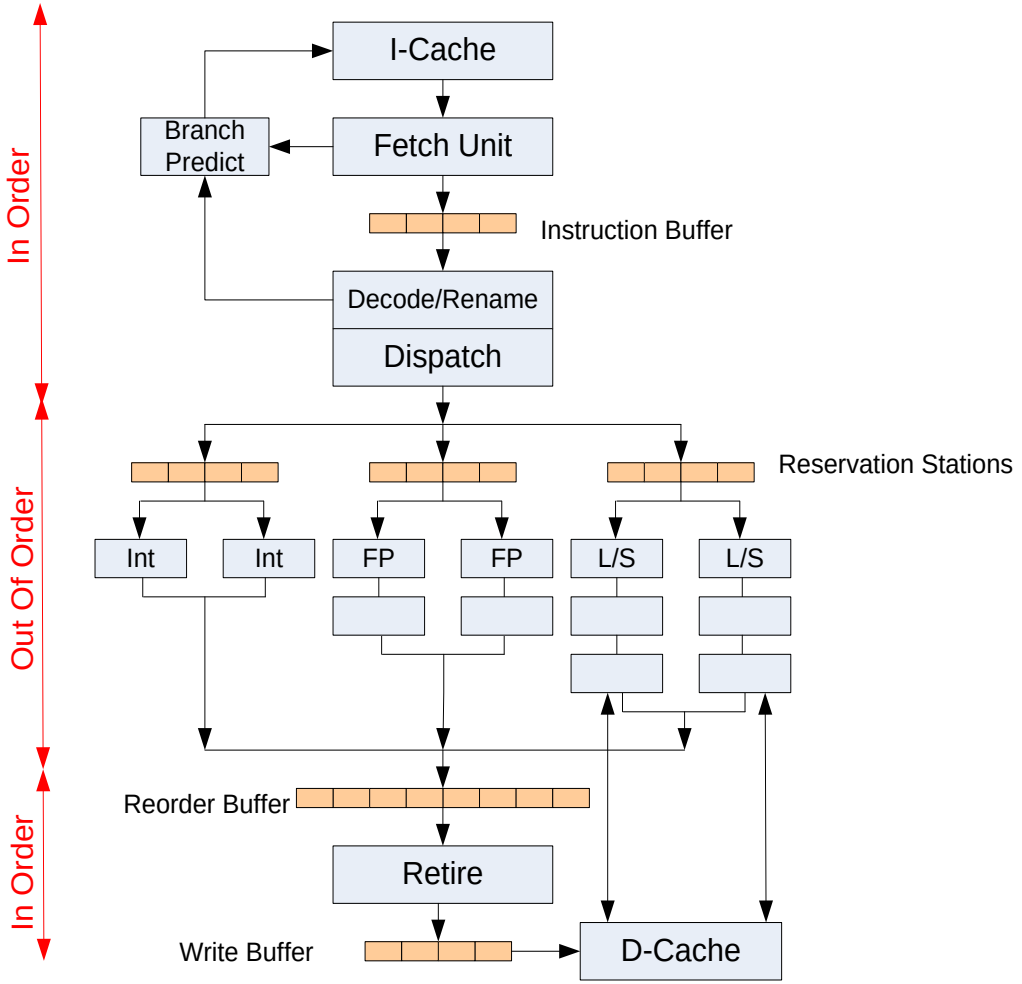


A Modern Out-of-Order Superscalar Processor



Reconstruct dataflow graph

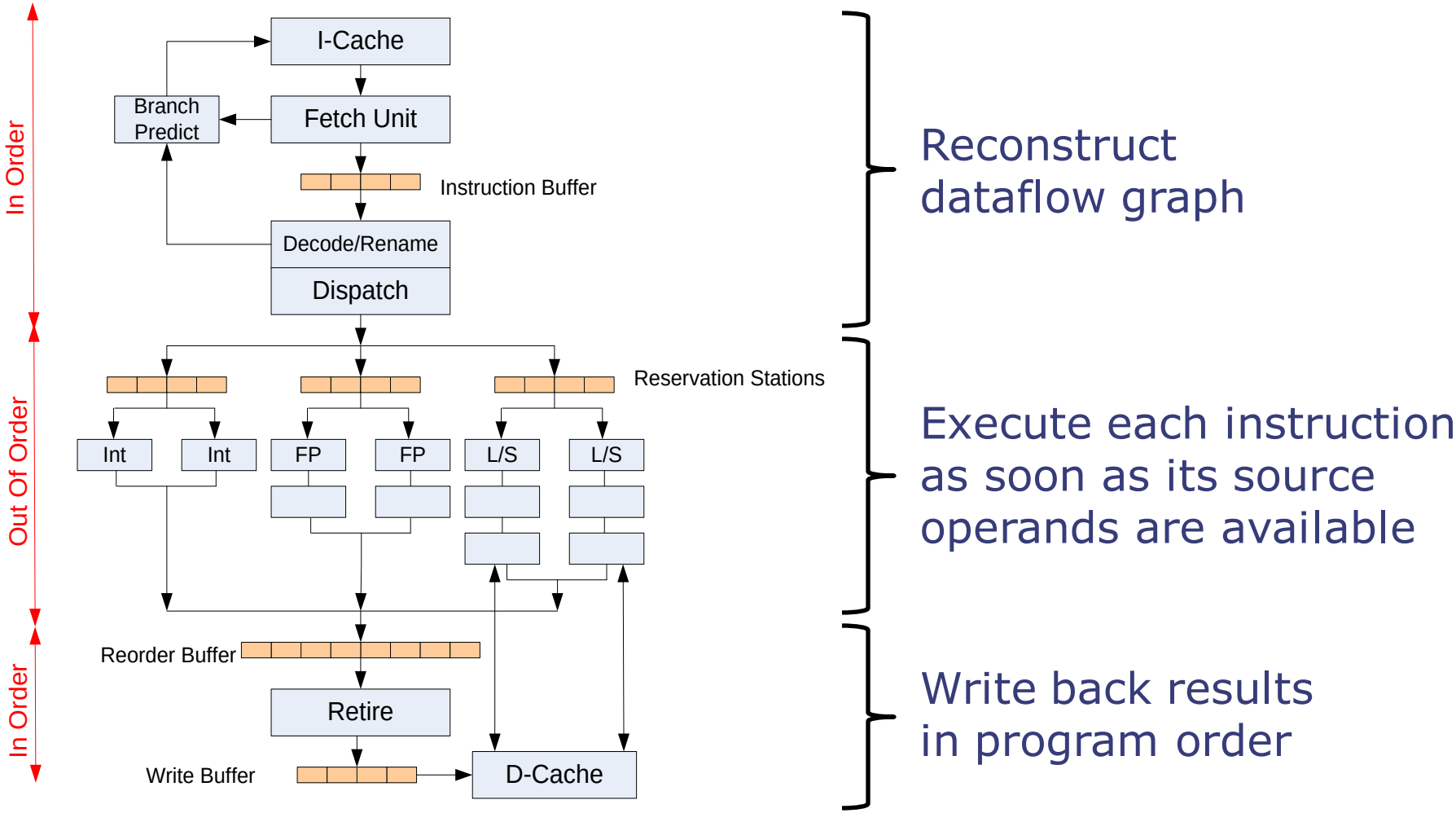
A Modern Out-of-Order Superscalar Processor



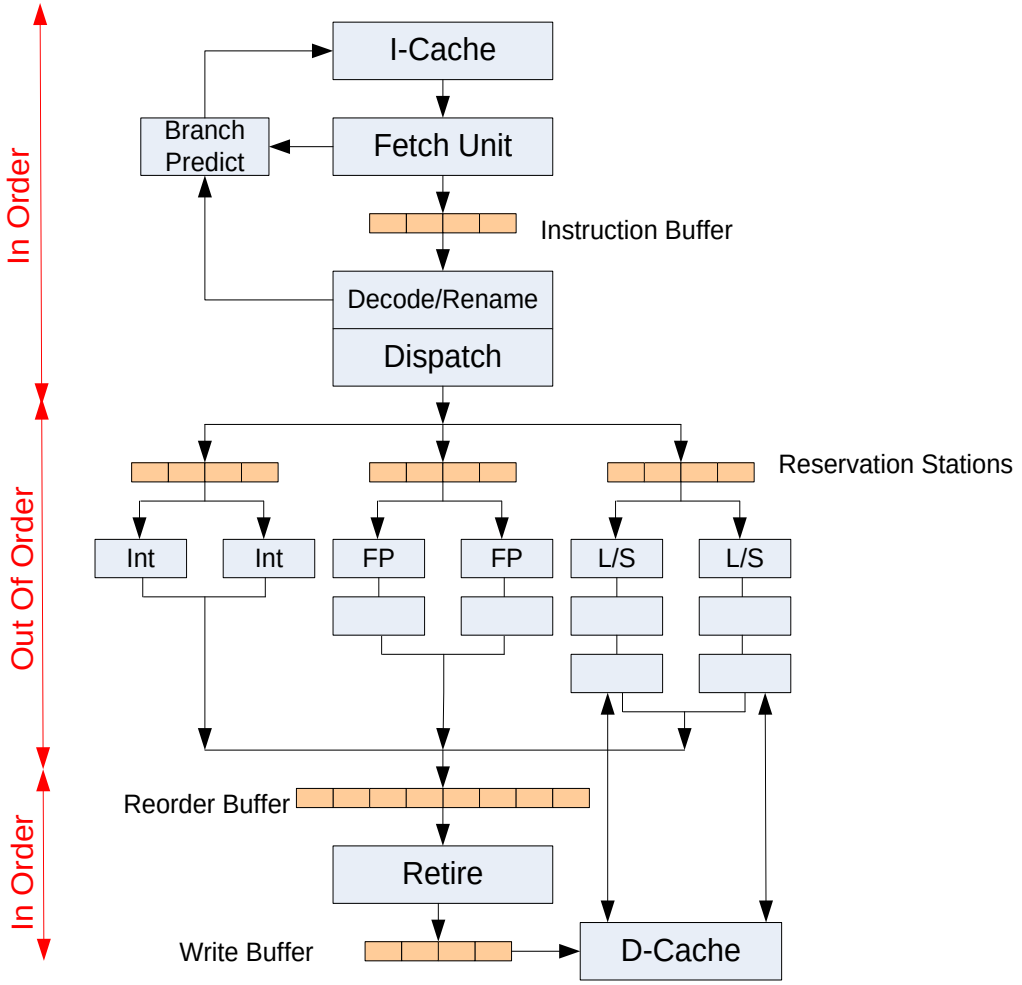
Reconstruct dataflow graph

Execute each instruction as soon as its source operands are available

A Modern Out-of-Order Superscalar Processor



A Modern Out-of-Order Superscalar Processor



Reconstruct dataflow graph

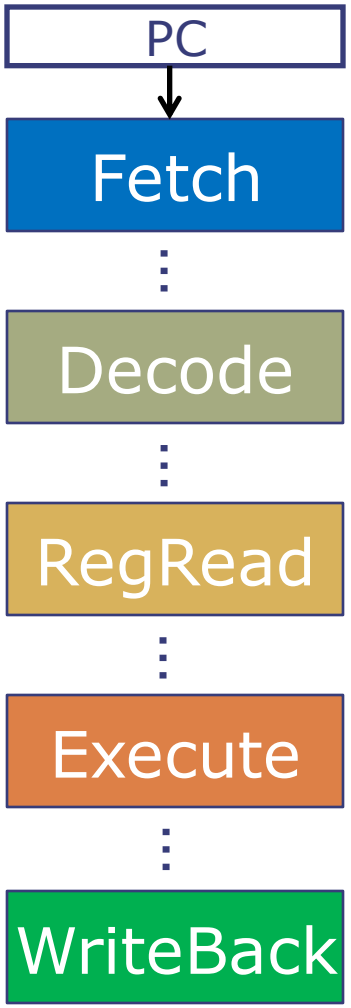
Execute each instruction as soon as its source operands are available

Write back results in program order

Why is this needed?

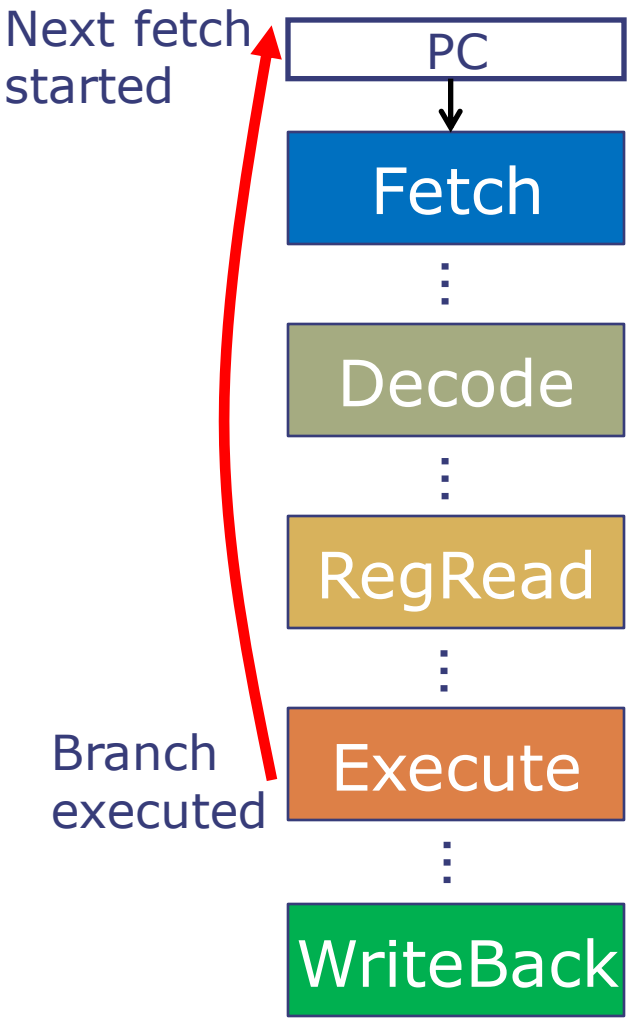
Control Hazard Penalty

- Modern processors have >10 pipeline stages between next PC calculation and branch resolution!



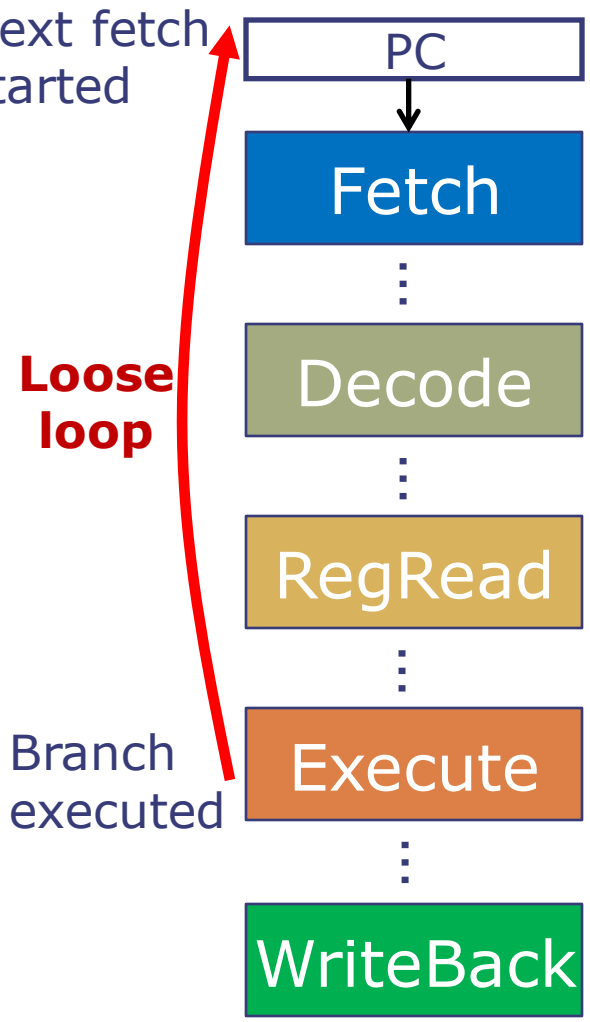
Control Hazard Penalty

- Modern processors have >10 pipeline stages between next PC calculation and branch resolution!



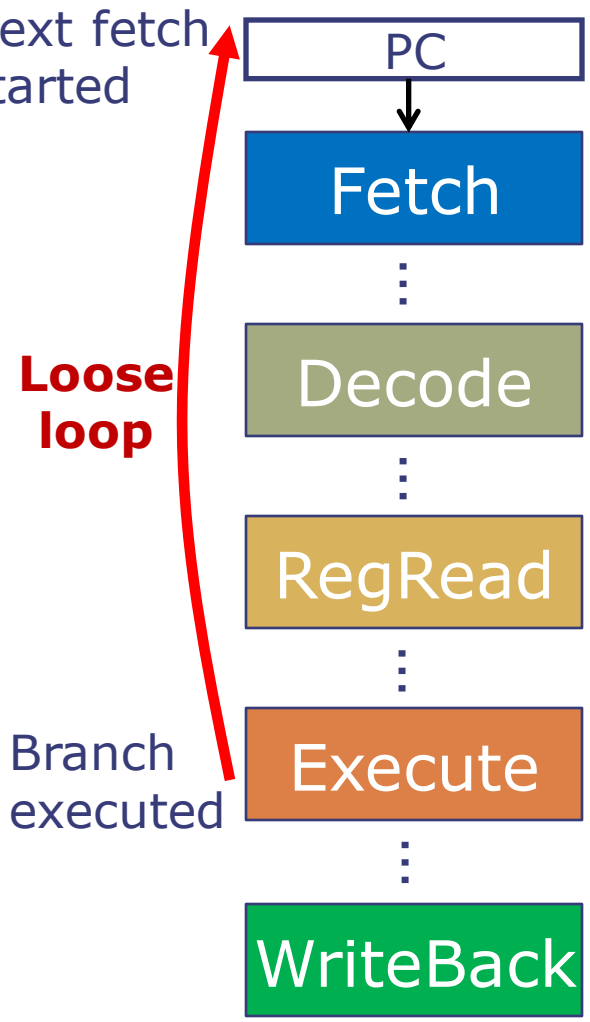
Control Hazard Penalty

- Modern processors have >10 pipeline stages between next PC calculation and branch resolution!



Control Hazard Penalty

- Modern processors have >10 pipeline stages between next PC calculation and branch resolution!
- How much work is lost every time pipeline does not follow correct instruction flow?

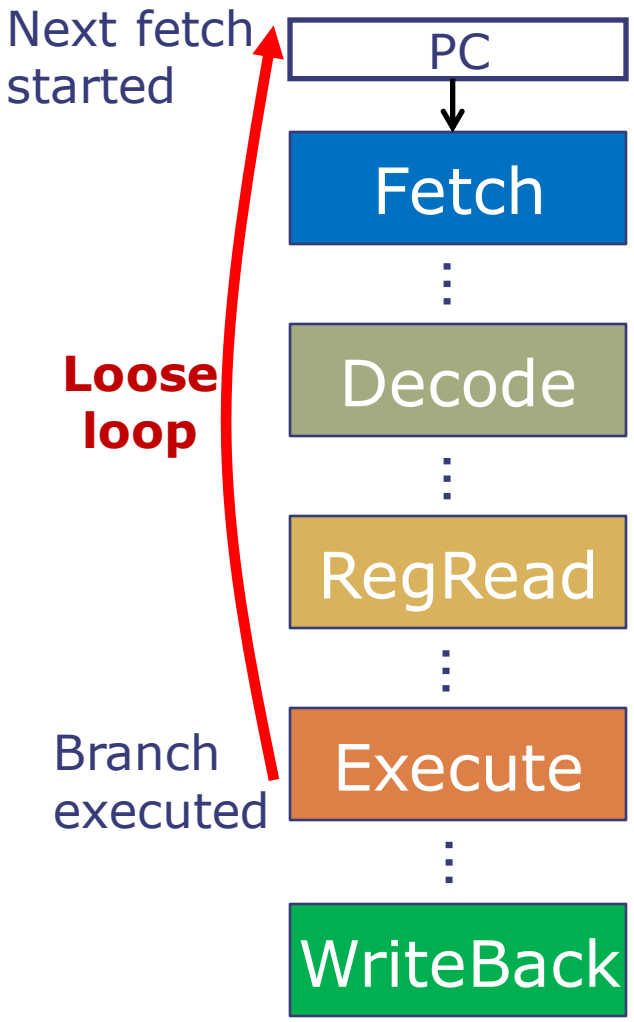


Control Hazard Penalty

- Modern processors have >10 pipeline stages between next PC calculation and branch resolution!

- How much work is lost every time pipeline does not follow correct instruction flow?

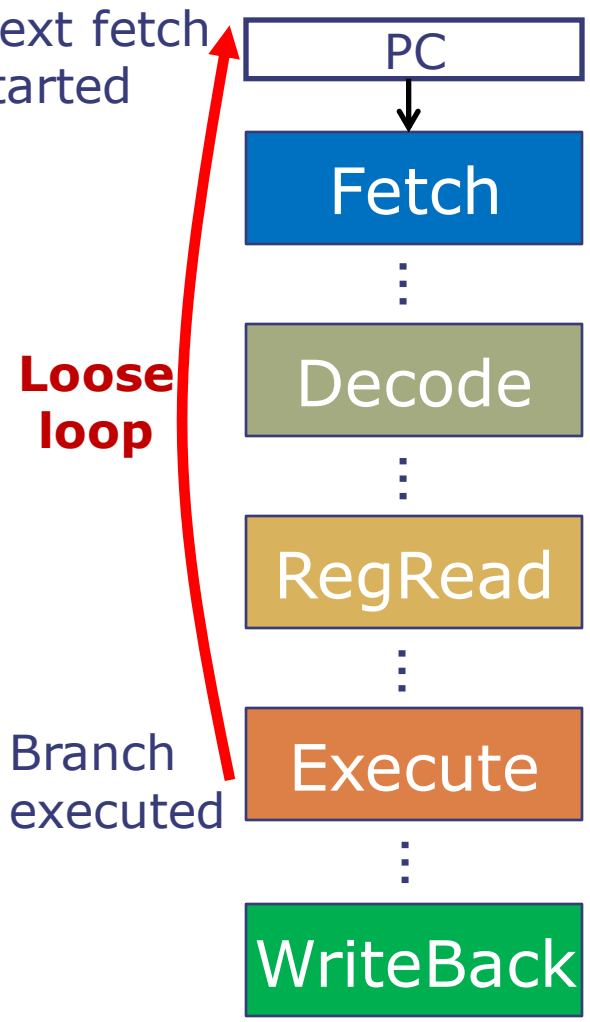
  Loop length x Pipeline width



Control Hazard Penalty

- Modern processors have >10 pipeline stages between next PC calculation and branch resolution!
- How much work is lost every time pipeline does not follow correct instruction flow?

  Loop length x Pipeline width
- One branch every 5-20 instructions… performance impact of mispredictions?



RISC-V Branches and Jumps

- Each instruction fetch depends on information from the preceding instruction:
  1) Is the preceding instruction a taken branch or jump?
  2) If so, what is the target address?

Instruction	*Taken known?*	*Target known?*
JAL		
JALR		
Branches		

RISC-V Branches and Jumps

- Each instruction fetch depends on information from the preceding instruction:
  1) Is the preceding instruction a taken branch or jump?
  2) If so, what is the target address?

Instruction	*Taken known?*	*Target known?*
JAL	After Inst. Decode	
JALR		
Branches		

RISC-V Branches and Jumps

- Each instruction fetch depends on information from the preceding instruction:
  1) Is the preceding instruction a taken branch or jump?
  2) If so, what is the target address?

Instruction	*Taken known?*	*Target known?*
JAL	After Inst. Decode	After Inst. Decode
JALR		
Branches		

RISC-V Branches and Jumps

- Each instruction fetch depends on information from the preceding instruction:
  1) Is the preceding instruction a taken branch or jump?
  2) If so, what is the target address?

Instruction	*Taken known?*	*Target known?*
JAL	After Inst. Decode	After Inst. Decode
JALR	After Inst. Decode	
Branches		

RISC-V Branches and Jumps

- Each instruction fetch depends on information from the preceding instruction:
  1) Is the preceding instruction a taken branch or jump?
  2) If so, what is the target address?

Instruction	*Taken known?*	*Target known?*
JAL	After Inst. Decode	After Inst. Decode
JALR	After Inst. Decode	After Inst. Execute
Branches		

RISC-V Branches and Jumps

- Each instruction fetch depends on information from the preceding instruction:
  1) Is the preceding instruction a taken branch or jump?
  2) If so, what is the target address?

Instruction	*Taken known?*	*Target known?*
JAL	After Inst. Decode	After Inst. Decode
JALR	After Inst. Decode	After Inst. Execute
Branches	After Inst. Execute	

RISC-V Branches and Jumps

- Each instruction fetch depends on information from the preceding instruction:
  1) Is the preceding instruction a taken branch or jump?
  2) If so, what is the target address?

Instruction	*Taken known?*	*Target known?*
JAL	After Inst. Decode	After Inst. Decode
JALR	After Inst. Decode	After Inst. Execute
Branches	After Inst. Execute	After Inst. Decode

Resolving Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages
- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated
- Strategy 3: Speculate
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value
- Strategy 4: Find something else to do

Resolving Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages
- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated
- Strategy 3: Speculate
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value
- Strategy 4: Find something else to do

Resolving Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages
- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated
- Strategy 3: Speculate
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value

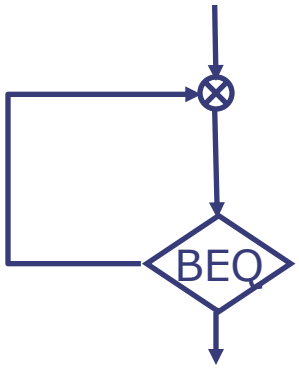
Predict jump/branch target and direction

- Strategy 4: Find something else to do

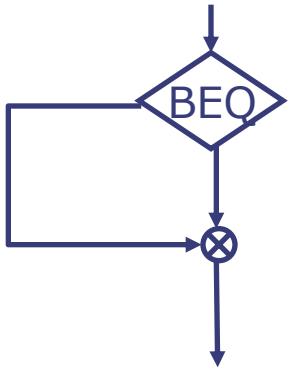
Static Branch Prediction

- Probability a branch is taken is ~60-70%, but:

backward
90%



forward
50%

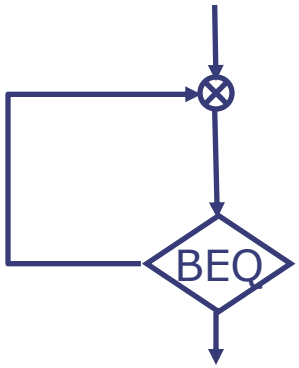


- Some ISAs attach preferred direction hints to branches, e.g., Motorola MC88110
  - bne0 *(preferred taken)* beq0 *(not taken)*
- Achieves ~80% accuracy

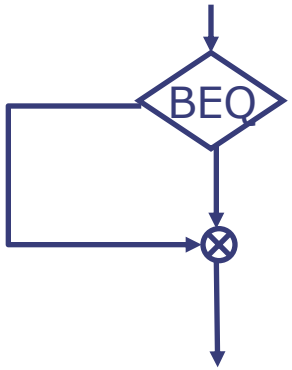
Static Branch Prediction

- Probability a branch is taken is ~60-70%, but:

backward
90%



forward
50%

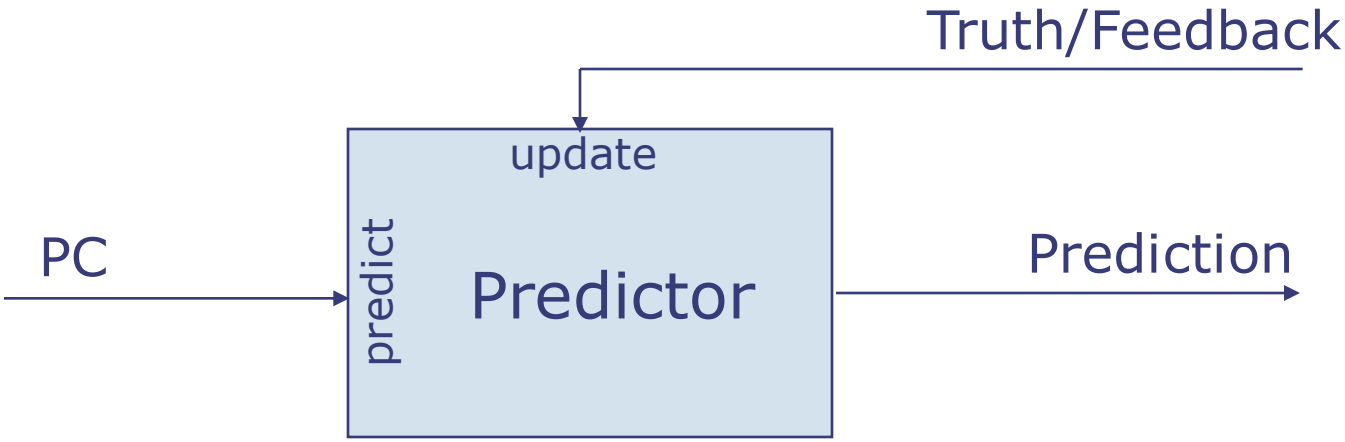


- Some ISAs attach preferred direction hints to branches, e.g., Motorola MC88110
  - bne0 *(preferred taken)* beq0 *(not taken)*
- Achieves ~80% accuracy

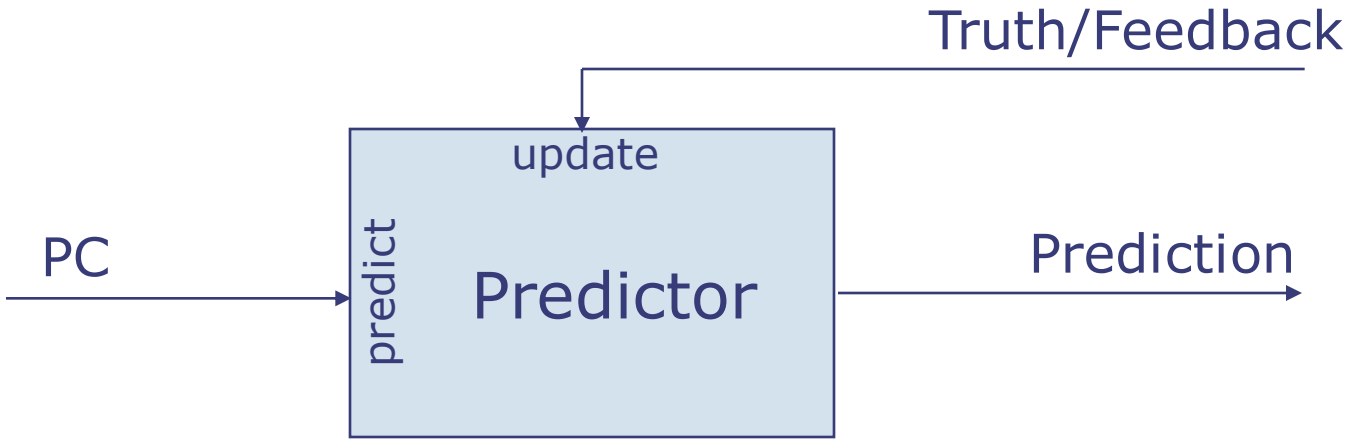
Good way to improve CPI on part 3 of the design project if you use a 4-stage pipeline

Dynamic Branch Prediction

Learning from past behavior

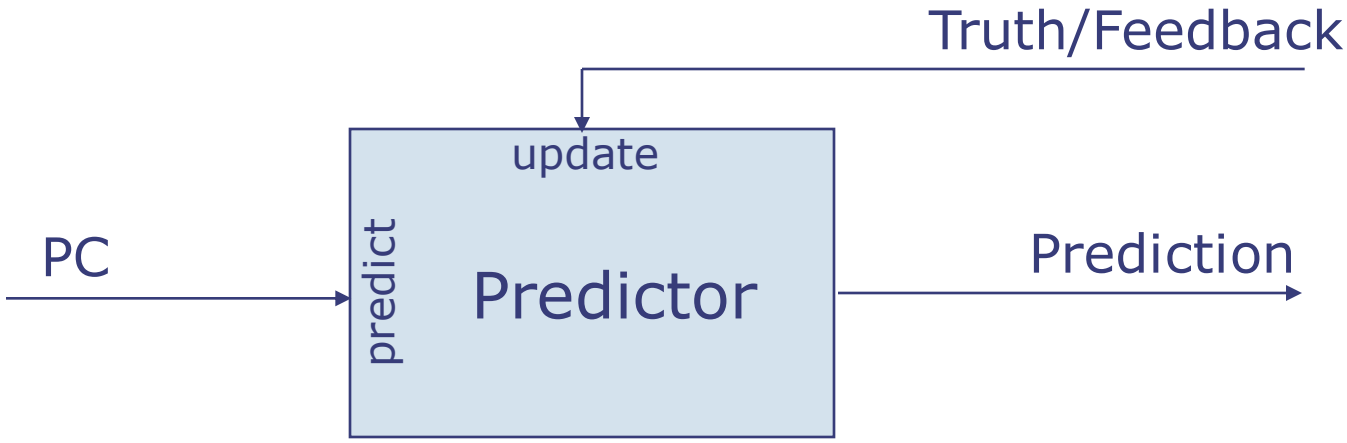


Dynamic Branch Prediction
Learning from past behavior



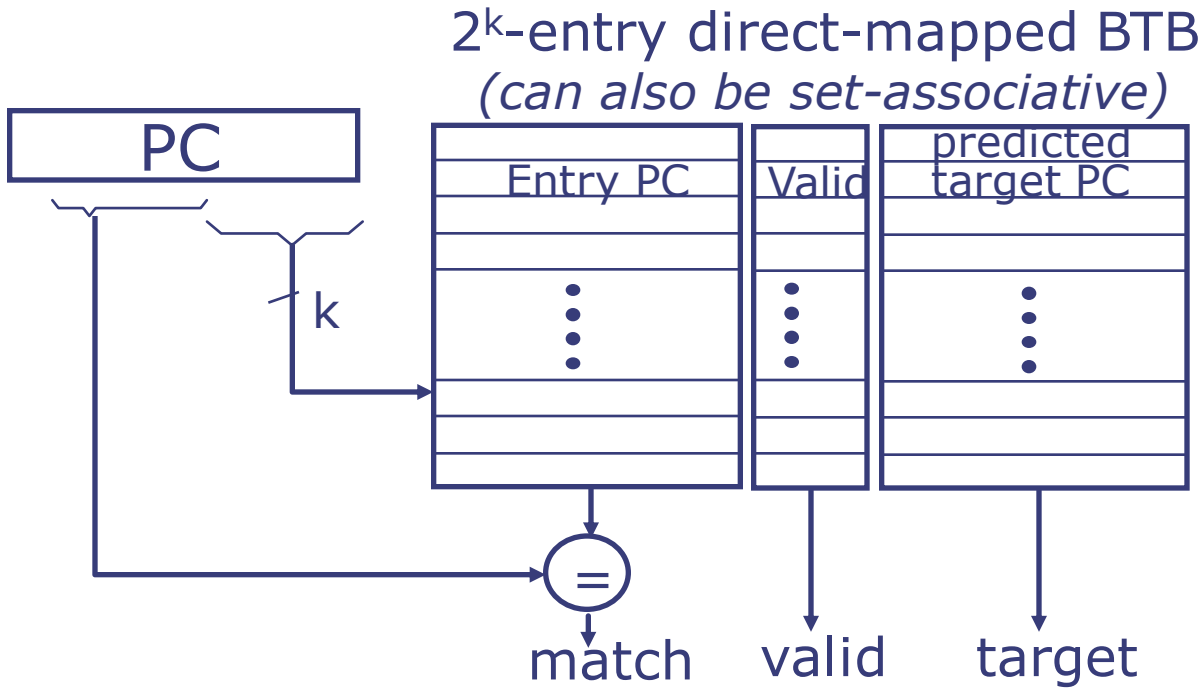
- Temporal correlation
  - The way a branch resolves may be a good predictor of the way it will resolve at the next execution

Dynamic Branch Prediction
Learning from past behavior



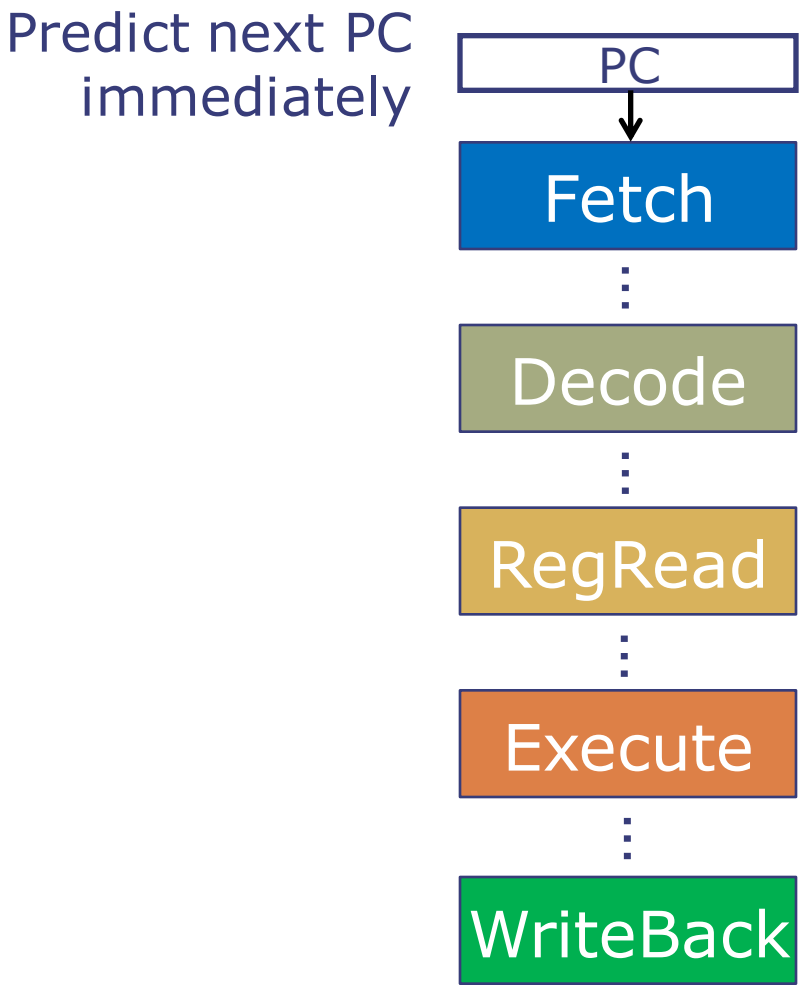
- Temporal correlation
  - The way a branch resolves may be a good predictor of the way it will resolve at the next execution
- Spatial correlation
  - Several branches may resolve in a highly correlated manner (a preferred path of execution)

Predicting the Target Address: Branch Target Buffer (BTB)

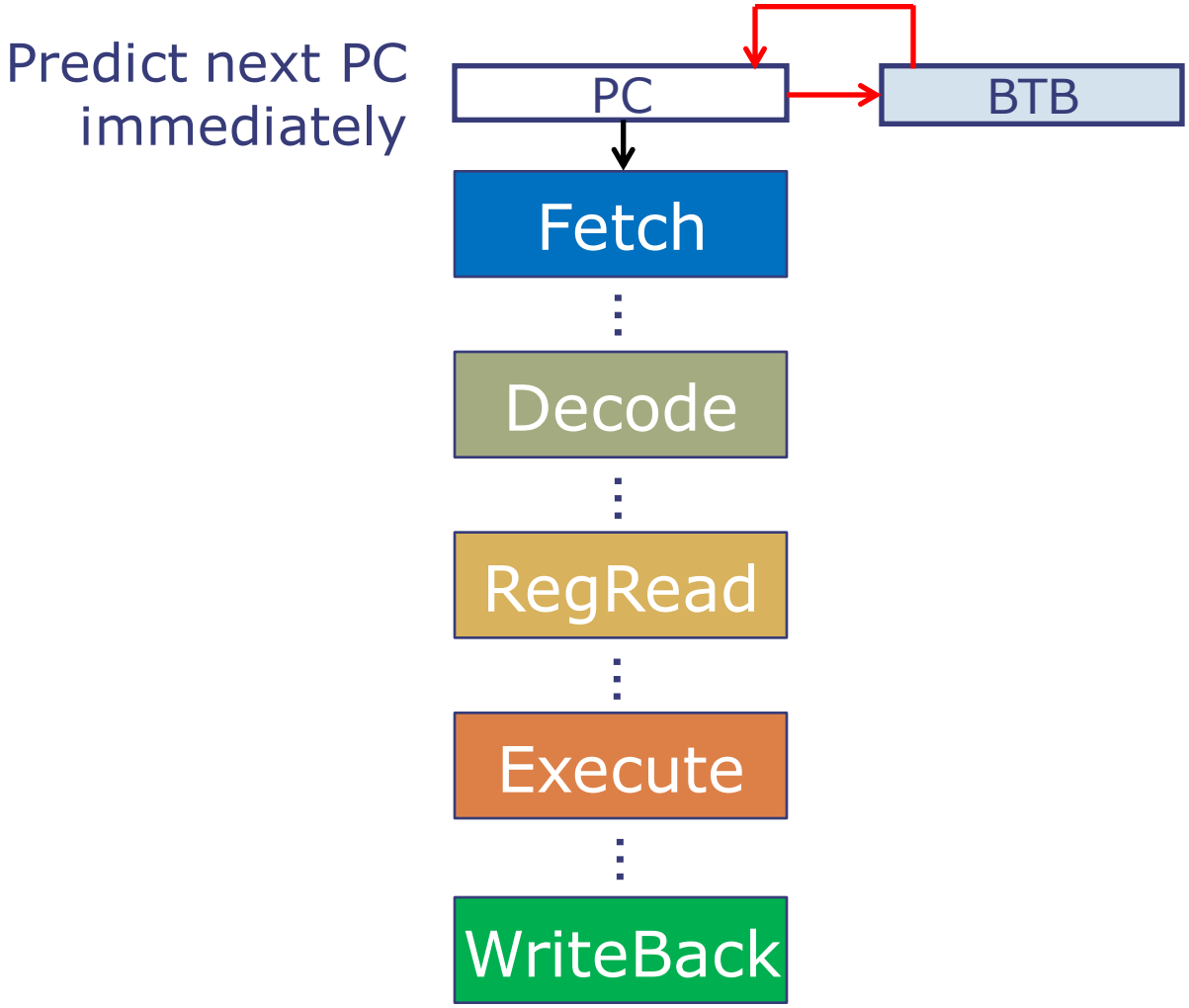


- BTB is a cache for targets: Remembers last target PC *for taken branches and jumps*
  - If hit, use stored target as predicted next PC
  - If miss, use PC+4 as predicted next PC
  - After target is known, update if prediction is wrong

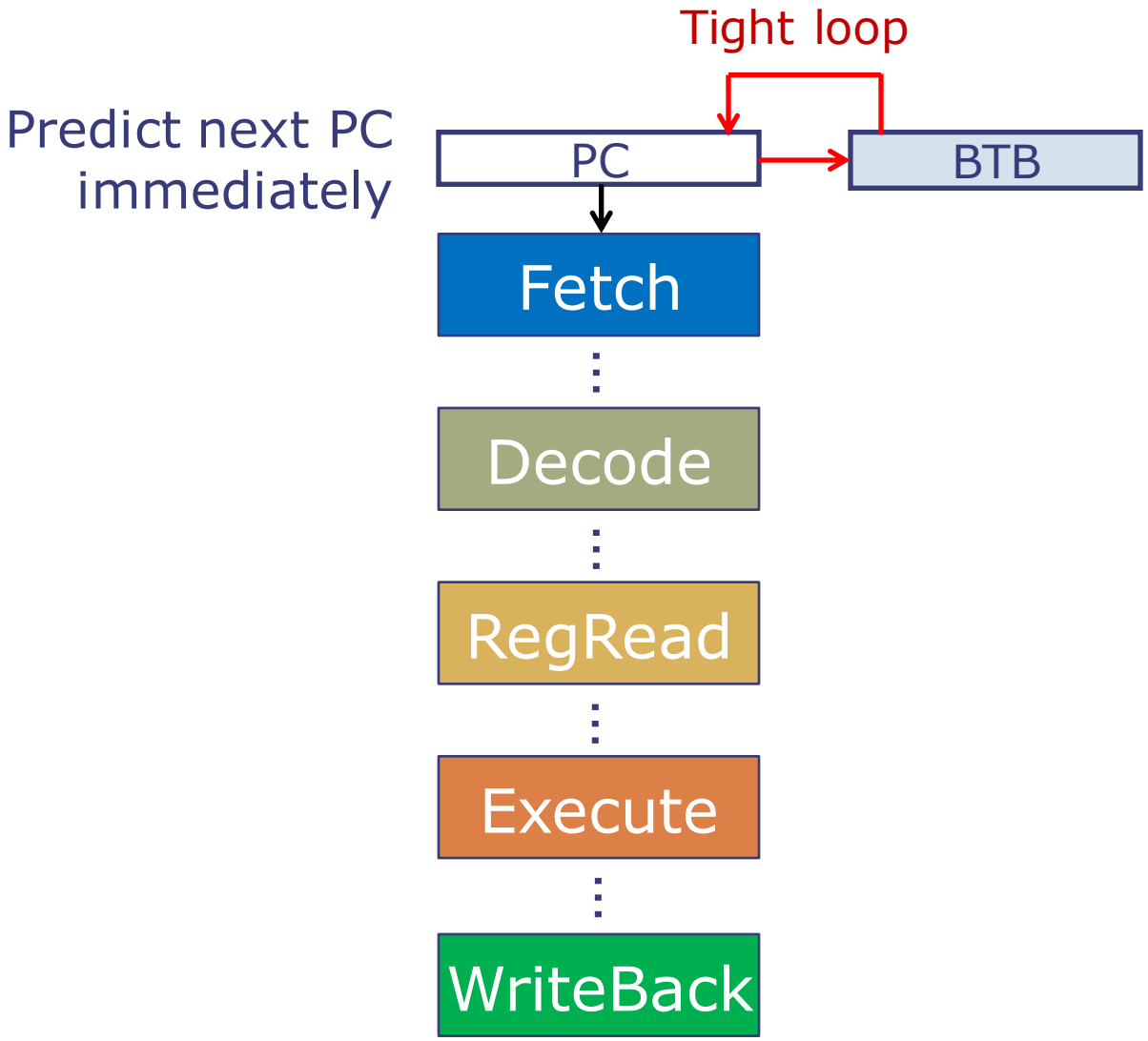
Integrating the BTB in the Pipeline



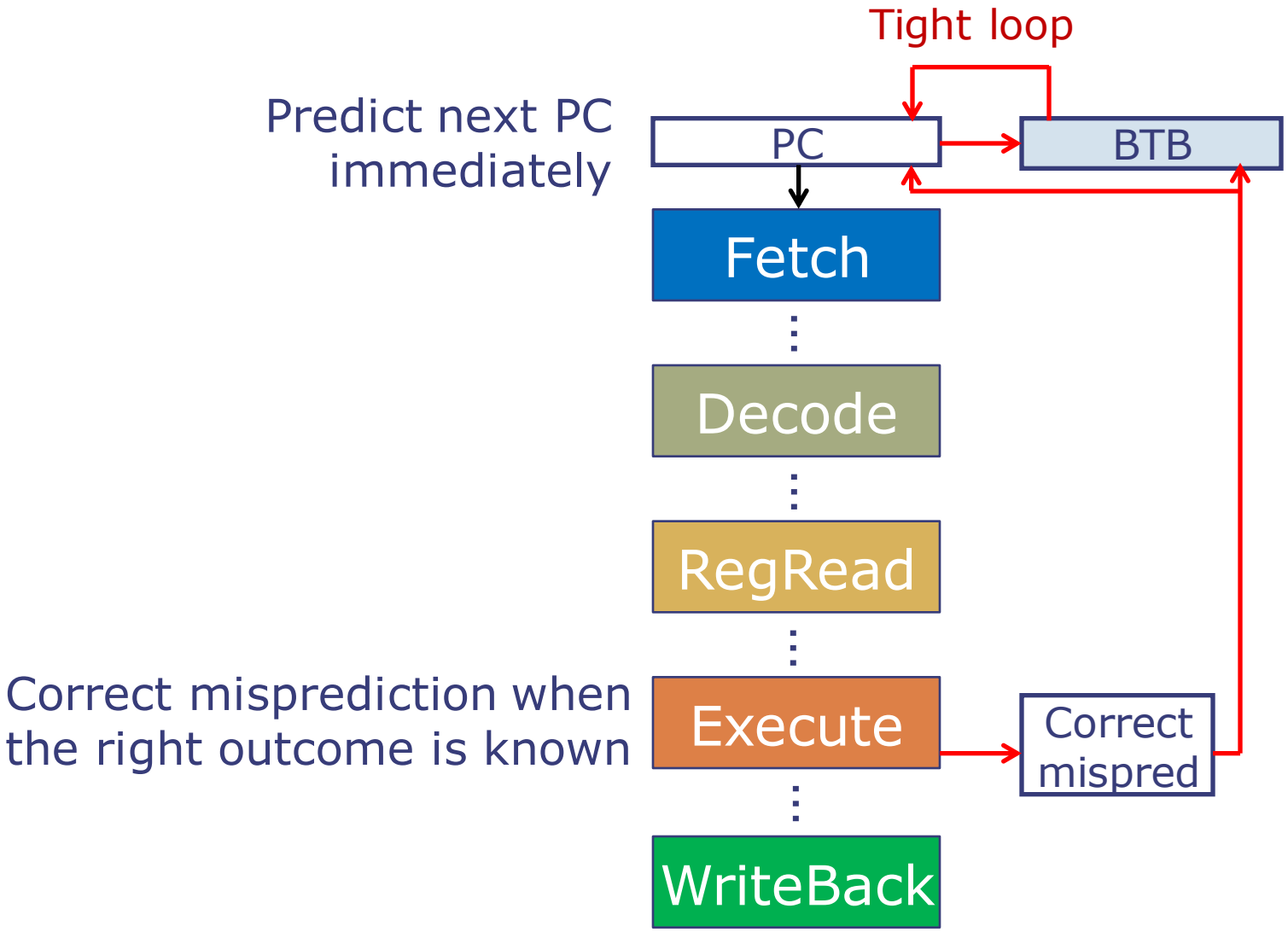
Integrating the BTB in the Pipeline



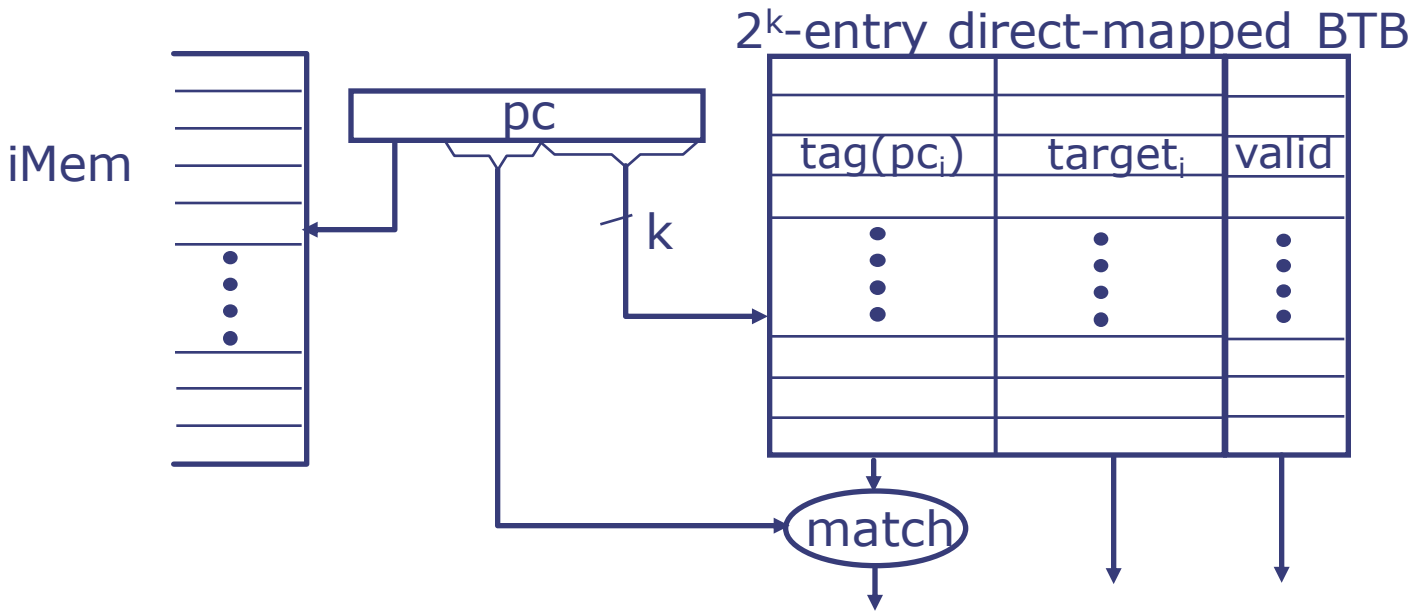
Integrating the BTB in the Pipeline



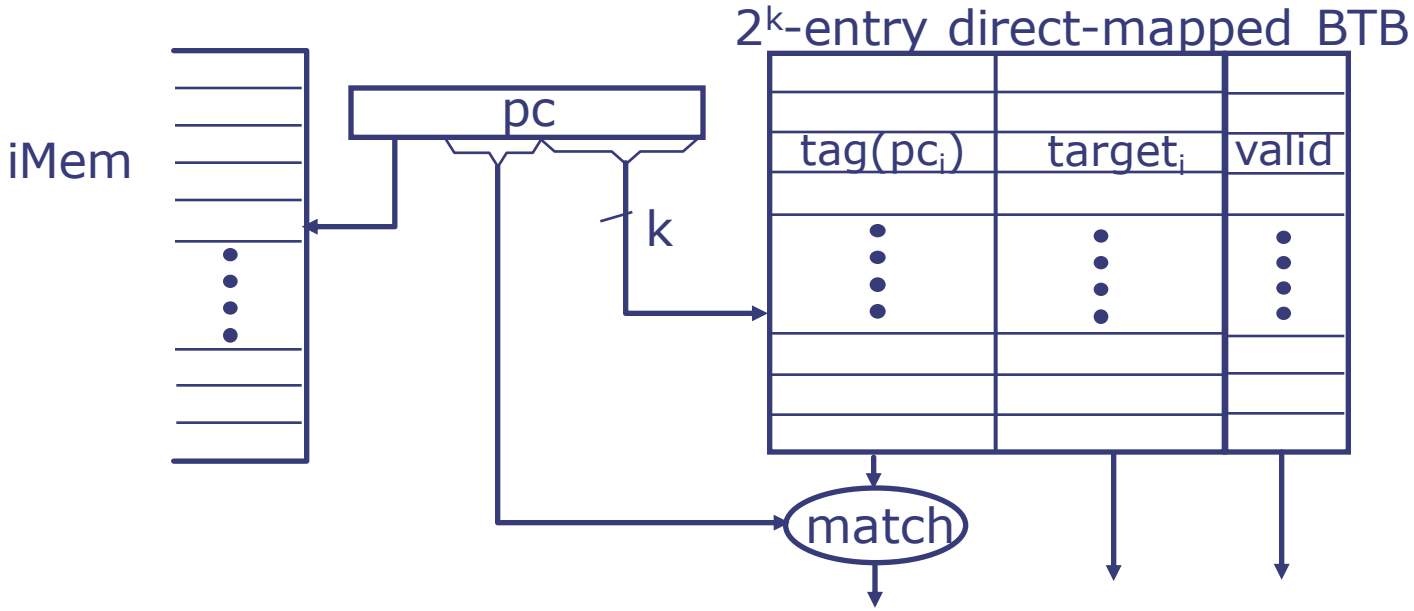
Integrating the BTB in the Pipeline



BTB Implementation Details

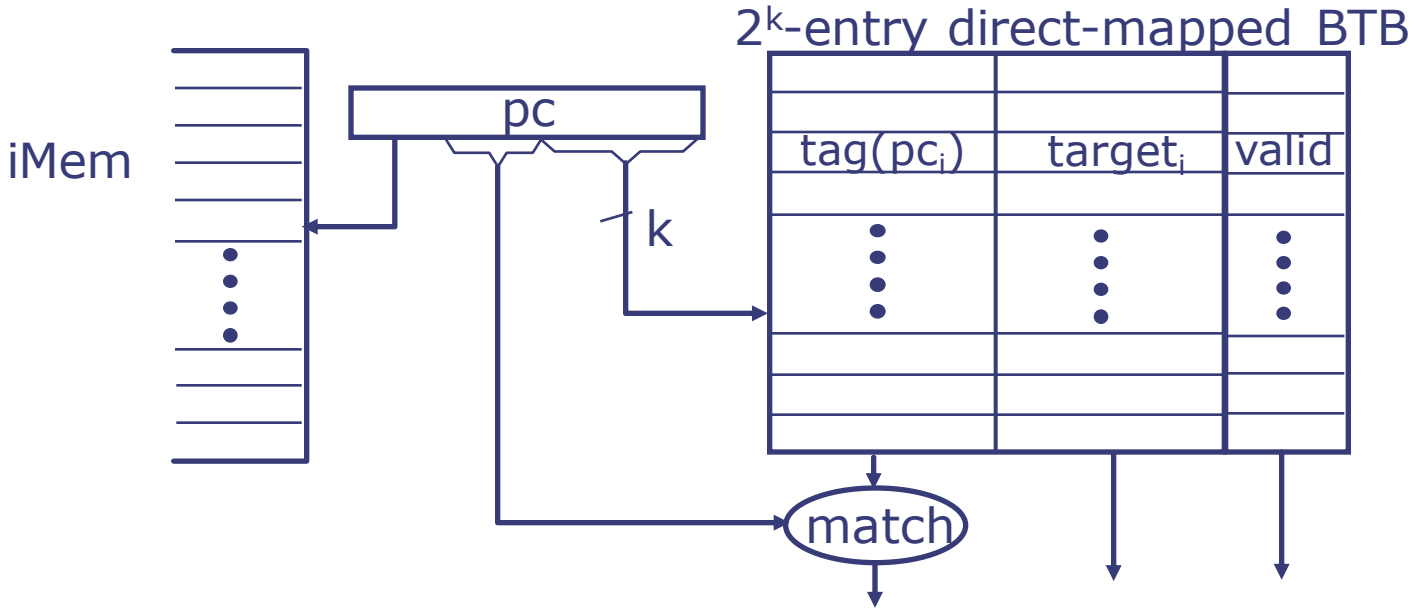


BTB Implementation Details



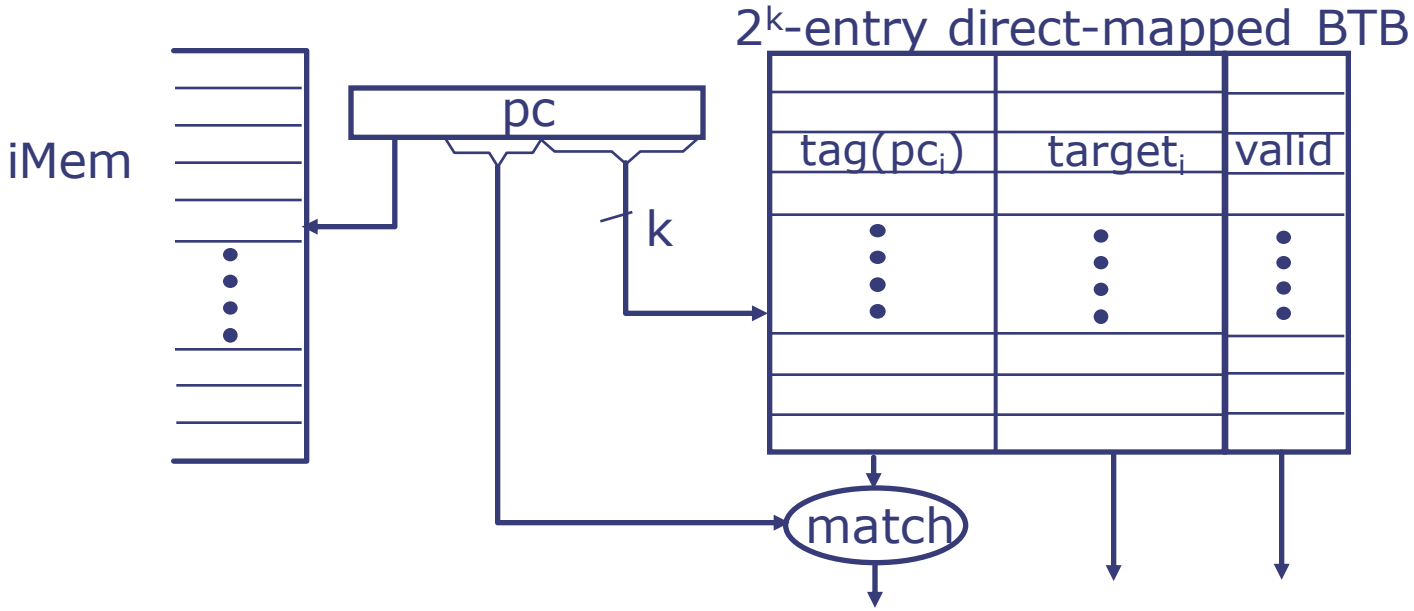
- Unlike caches, it is fine if the BTB produces an invalid next PC
  - It's just a prediction!

BTB Implementation Details



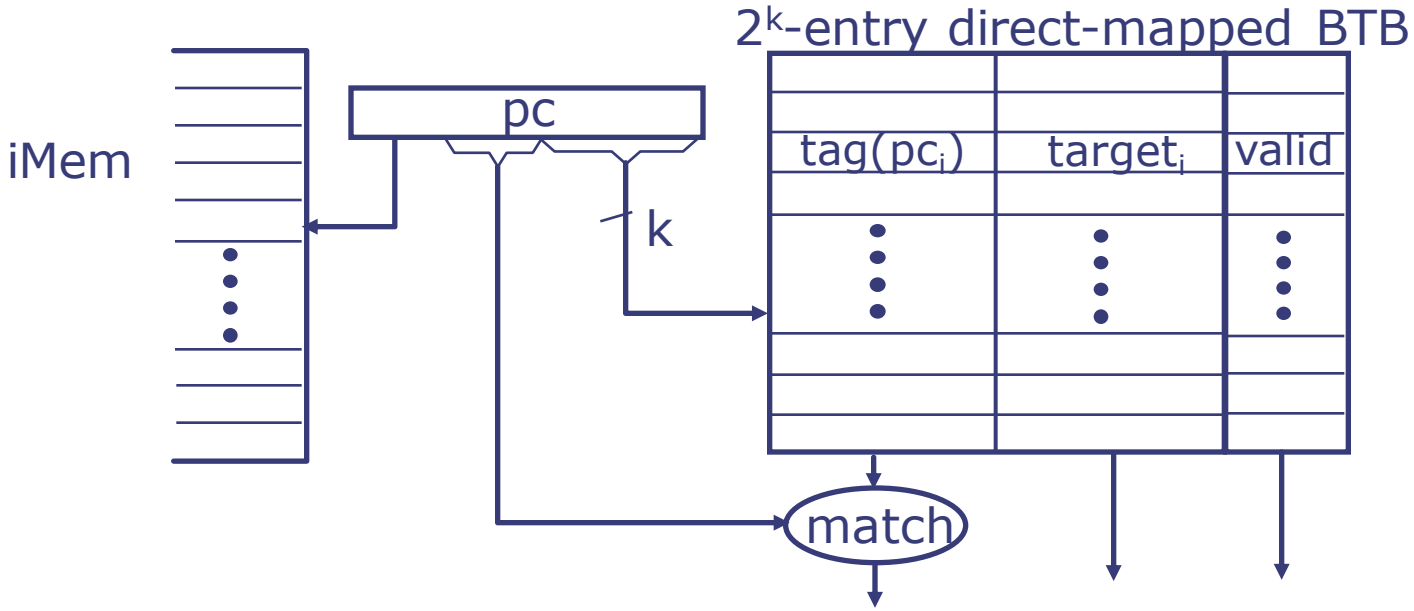
- Unlike caches, it is fine if the BTB produces an invalid next PC
  - It's just a prediction!
- Therefore, BTB area & delay can be reduced by

BTB Implementation Details



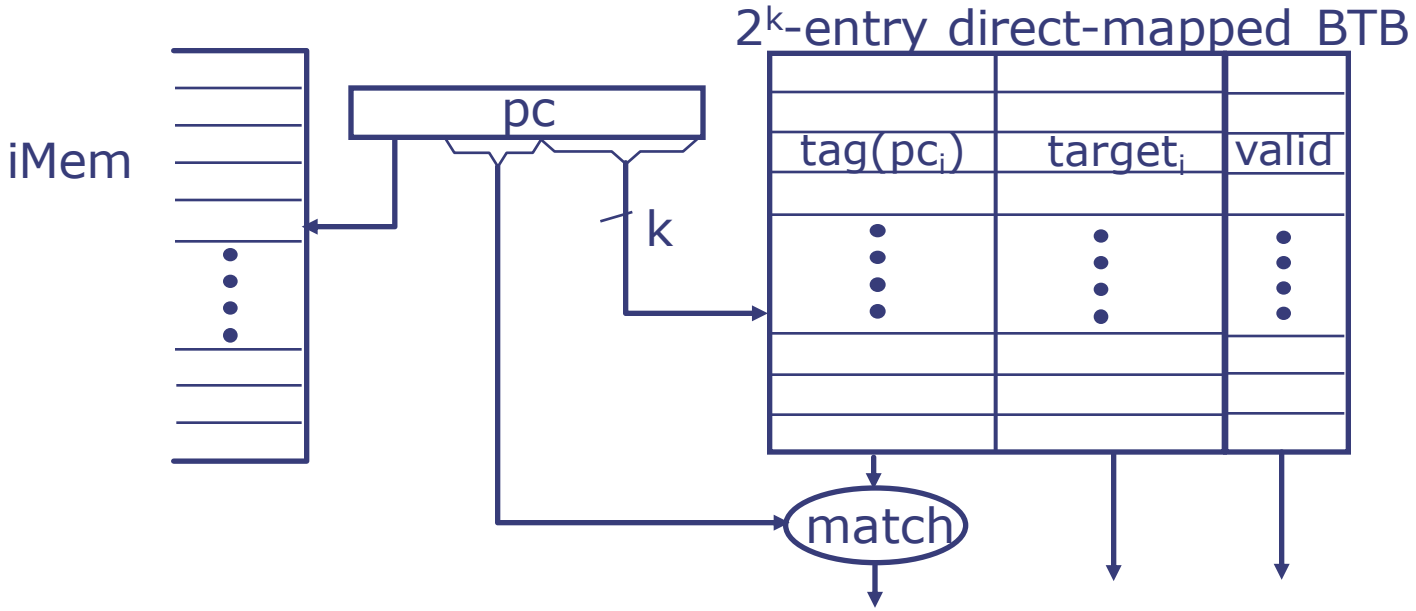
- Unlike caches, it is fine if the BTB produces an invalid next PC
  - It's just a prediction!
- Therefore, BTB area & delay can be reduced by
  - Making tags arbitrarily small (match with a subset of PC bits)

BTB Implementation Details



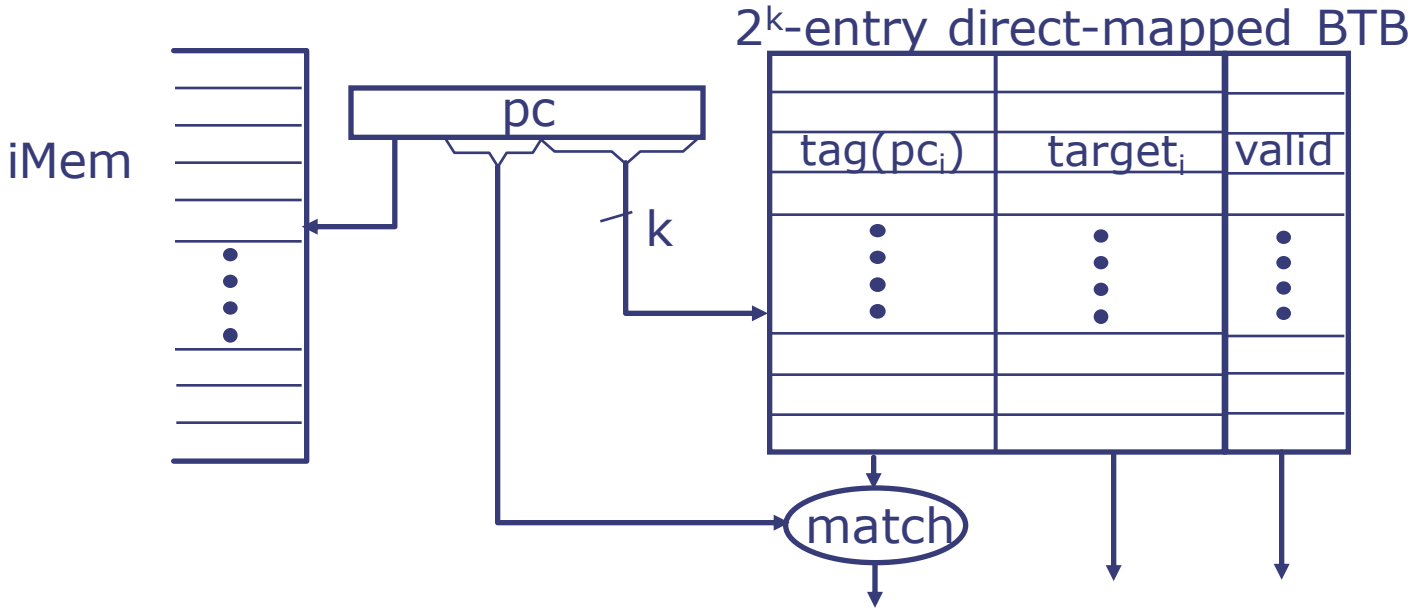
- Unlike caches, it is fine if the BTB produces an invalid next PC
  - It's just a prediction!
- Therefore, BTB area & delay can be reduced by
  - Making tags arbitrarily small (match with a subset of PC bits)
  - Storing only a subset of target PC bits (fill missing bits from current PC)

BTB Implementation Details



- Unlike caches, it is fine if the BTB produces an invalid next PC
  - It's just a prediction!
- Therefore, BTB area & delay can be reduced by
  - Making tags arbitrarily small (match with a subset of PC bits)
  - Storing only a subset of target PC bits (fill missing bits from current PC)
  - Not storing valid bits

BTB Implementation Details



- Unlike caches, it is fine if the BTB produces an invalid next PC
  - It's just a prediction!
- Therefore, BTB area & delay can be reduced by
  - Making tags arbitrarily small (match with a subset of PC bits)
  - Storing only a subset of target PC bits (fill missing bits from current PC)
  - Not storing valid bits
- Even small BTBs are very effective!

BTB Interface

```
typedef struct
  { Word pc; Word nextPc; Bool taken; } UpdateArgs;
module BTB;
  method Addr predict(Addr pc);
  input Maybe#(UpdateArgs) update default = Invalid;
endmodule
```

BTB Interface

```
typedef struct
  { Word pc; Word nextPc; Bool taken; } UpdateArgs;
module BTB;
  method Addr predict(Addr pc);
  input Maybe#(UpdateArgs) update default = Invalid;
endmodule
```

- *predict:* Simple lookup to predict nextPC in Fetch stage

BTB Interface

```
typedef struct
  { Word pc; Word nextPc; Bool taken; } UpdateArgs;
module BTB;
  method Addr predict(Addr pc);
  input Maybe#(UpdateArgs) update default = Invalid;
endmodule
```

- *predict:* Simple lookup to predict nextPC in Fetch stage
- *update:* On a pc misprediction, if the jump or branch at the pc was taken, then the BTB is updated with the new (pc, nextPC). Otherwise, the pc entry is deleted.

BTB Interface

```
typedef struct
  { Word pc; Word nextPc; Bool taken; } UpdateArgs;
module BTB;
  method Addr predict(Addr pc);
  input Maybe#(UpdateArgs) update default = Invalid;
endmodule
```

- *predict:* Simple lookup to predict nextPC in Fetch stage
- *update:* On a pc misprediction, if the jump or branch at the pc was taken, then the BTB is updated with the new (pc, nextPC). Otherwise, the pc entry is deleted.

A BTB is a good way to improve CPI
on part 3 of the design project
(and has lower t_{CLK} than static prediction)

Better Branch Direction Prediction

- Consider the following loop:

```
loop: …
      addi a1, a1, -1
      bnez a1, loop
```

Better Branch Direction Prediction

- Consider the following loop:

```
loop: …
      addi a1, a1, -1
      bnez a1, loop
```

- *How many mispredictions does the BTB incur per loop?*

Better Branch Direction Prediction

- Consider the following loop:

```
loop: …
      addi a1, a1, -1
      bnez a1, loop
```

- *How many mispredictions does the BTB incur per loop?*
  - *One on loop exit*

Better Branch Direction Prediction

- Consider the following loop:

```
loop: …
      addi a1, a1, -1
      bnez a1, loop
```

- *How many mispredictions does the BTB incur per loop?*
  - *One on loop exit*
  - *Another one on first iteration*

Two-Bit Direction Predictor
Smith 1981

- Use two bits per BTB entry instead of one valid bit
- Manage them as a saturating counter:

On not-taken ↓	↑ On taken			
		1	1	Strongly taken
		1	0	Weakly taken
		0	1	Weakly not-taken
		0	0	Strongly not-taken

  - Direction prediction changes only after two wrong predictions

Two-Bit Direction Predictor
Smith 1981

- Use two bits per BTB entry instead of one valid bit
- Manage them as a saturating counter:

On not-taken ↓	↑ On taken			
		1	1	Strongly taken
		1	0	Weakly taken
		0	1	Weakly not-taken
		0	0	Strongly not-taken

  - Direction prediction changes only after two wrong predictions

- *How many mispredictions per loop?* ___

Two-Bit Direction Predictor
Smith 1981

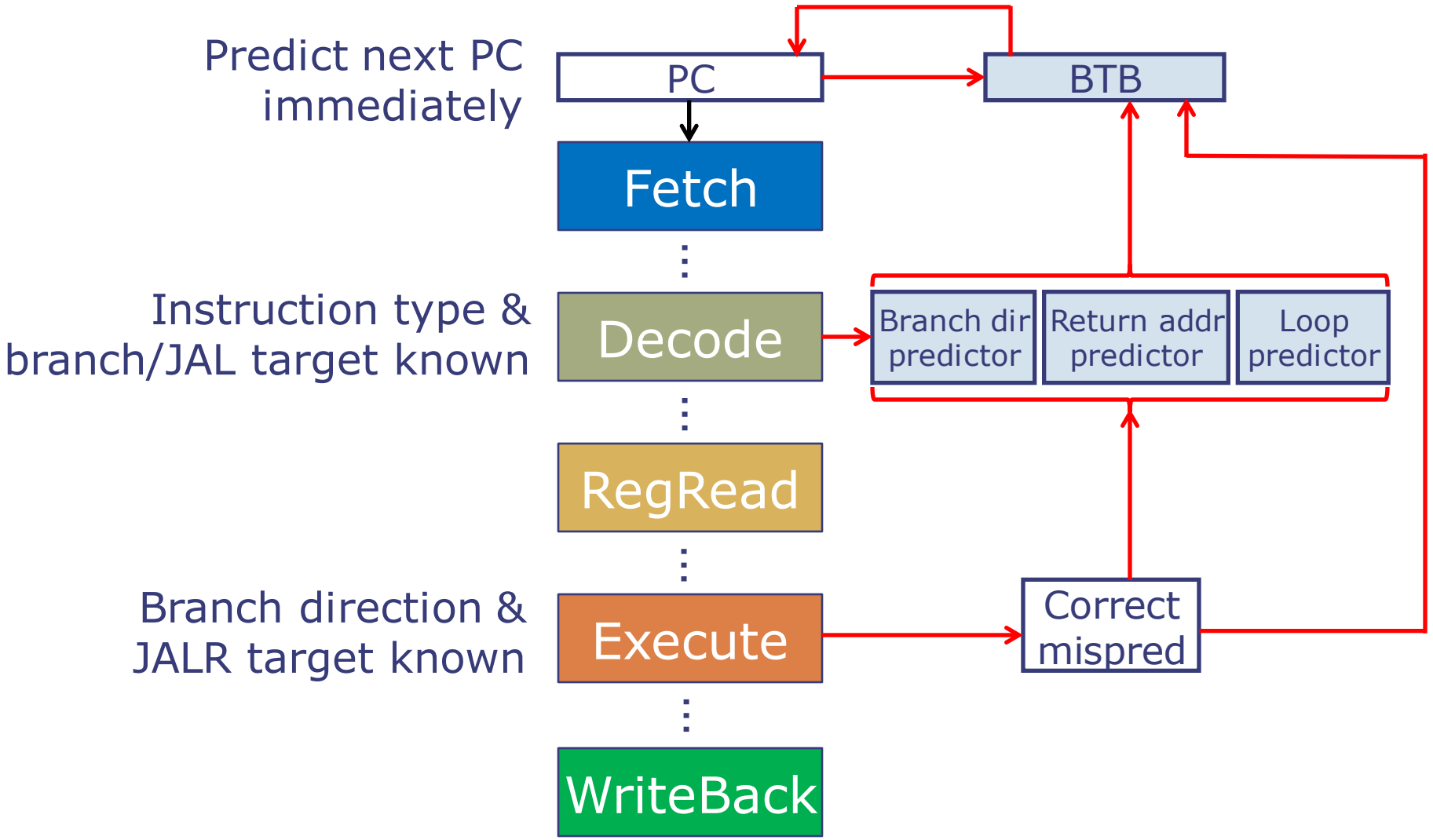
- Use two bits per BTB entry instead of one valid bit
- Manage them as a saturating counter:

On not-taken ↓	↑ On taken			
		1	1	Strongly taken
		1	0	Weakly taken
		0	1	Weakly not-taken
		0	0	Strongly not-taken

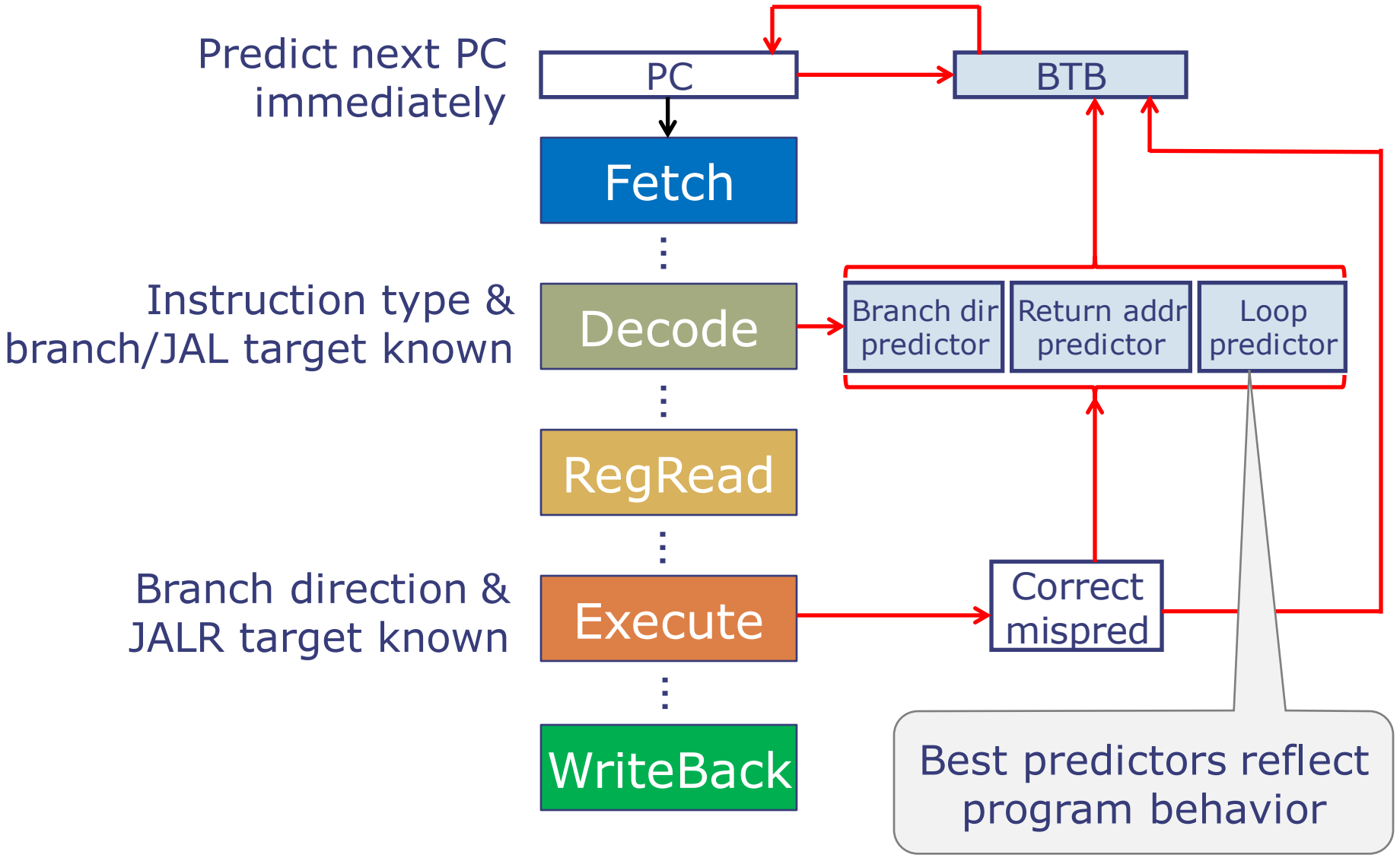
  - Direction prediction changes only after two wrong predictions

- *How many mispredictions per loop?* 1

Modern Processors Combine Multiple Specialized Predictors

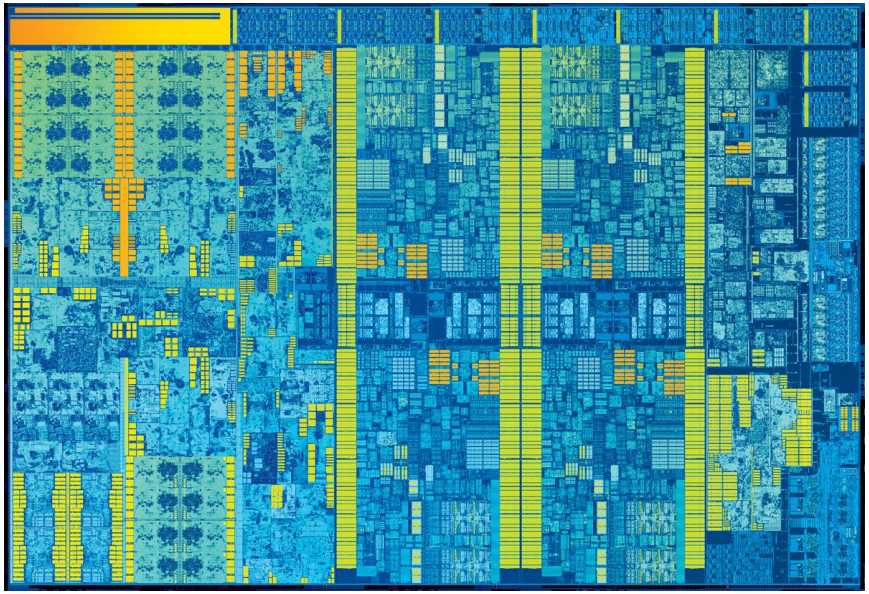


Modern Processors Combine Multiple Specialized Predictors



Putting It All Together: Intel Core i7 (Skylake)

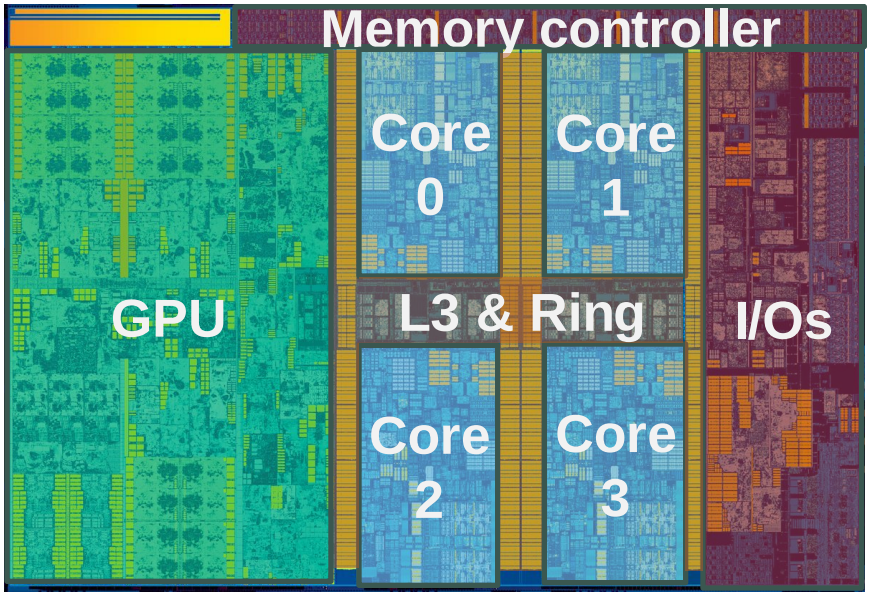
- Each core has 19 pipeline stages, ~4GHz
- 6-wide superscalar
- Out of order execution
- Multi-level branch predictors
- Caches:
  - L1: 32KB I + 32KB D
  - L2: 256KB
  - L3: 8MB, shared



Intel, 2016, 14nm,
1.7B transistors, 122mm^2

Putting It All Together: Intel Core i7 (Skylake)

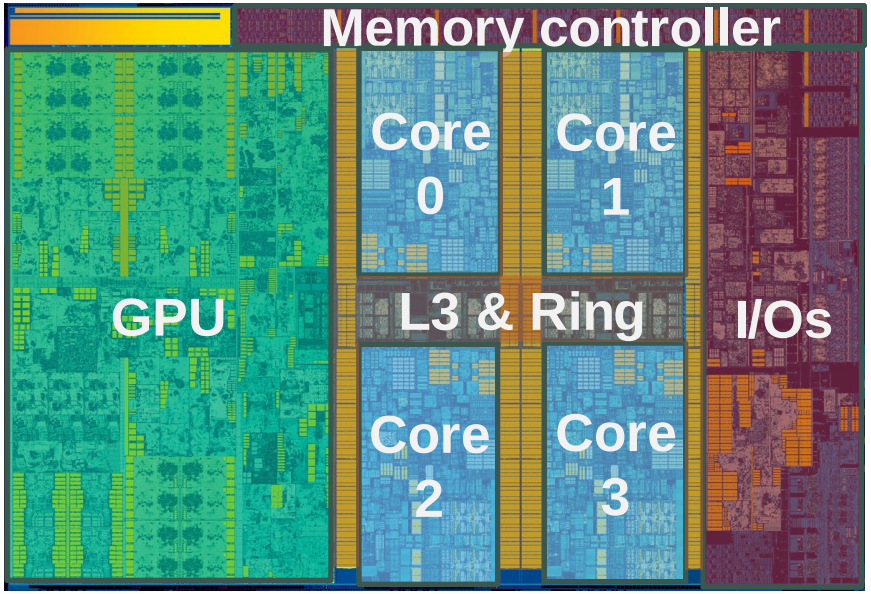
- Each core has 19 pipeline stages, ~4GHz
- 6-wide superscalar
- Out of order execution
- Multi-level branch predictors
- Caches:
  - L1: 32KB I + 32KB D
  - L2: 256KB
  - L3: 8MB, shared



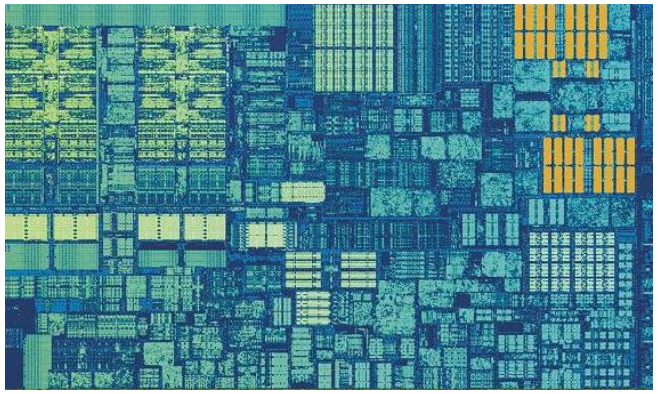
Intel, 2016, 14nm,
1.7B transistors, 122mm^2

Putting It All Together: Intel Core i7 (Skylake)

- Each core has 19 pipeline stages, ~4GHz
- 6-wide superscalar
- Out of order execution
- Multi-level branch predictors
- Caches:
  - L1: 32KB I + 32KB D
  - L2: 256KB
  - L3: 8MB, shared

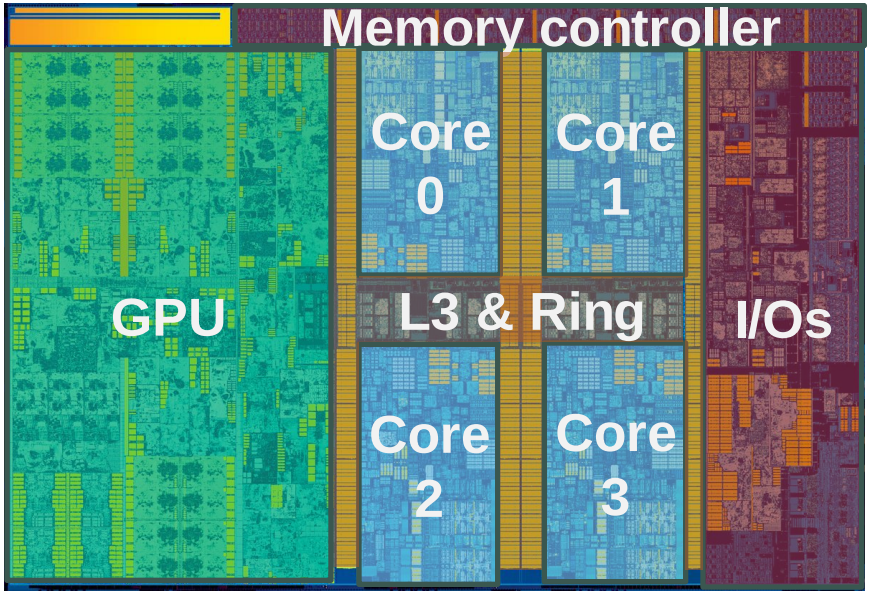


Intel, 2016, 14nm,
1.7B transistors, 122mm^2

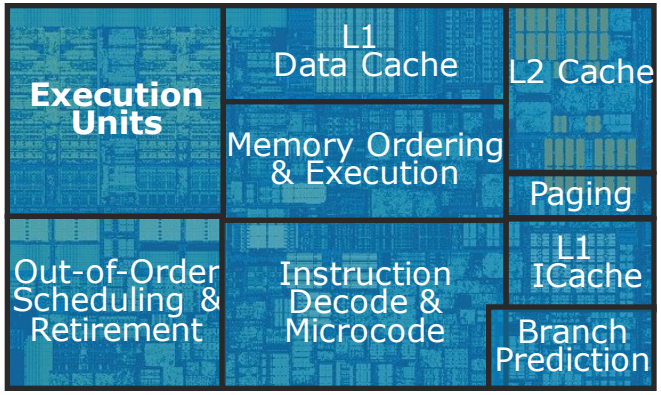


Putting It All Together: Intel Core i7 (Skylake)

- Each core has 19 pipeline stages, ~4GHz
- 6-wide superscalar
- Out of order execution
- Multi-level branch predictors
- Caches:
  - L1: 32KB I + 32KB D
  - L2: 256KB
  - L3: 8MB, shared

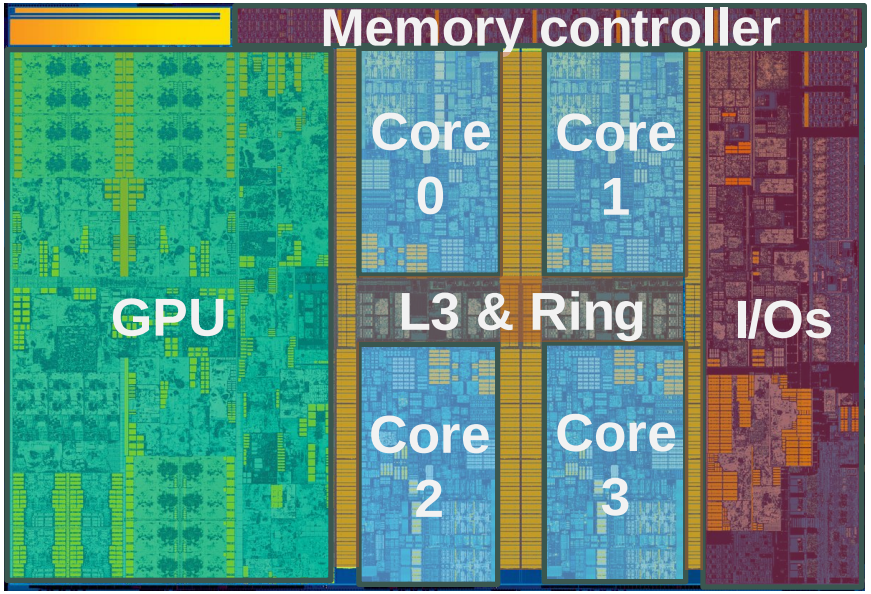


Intel, 2016, 14nm,
1.7B transistors, 122mm^2

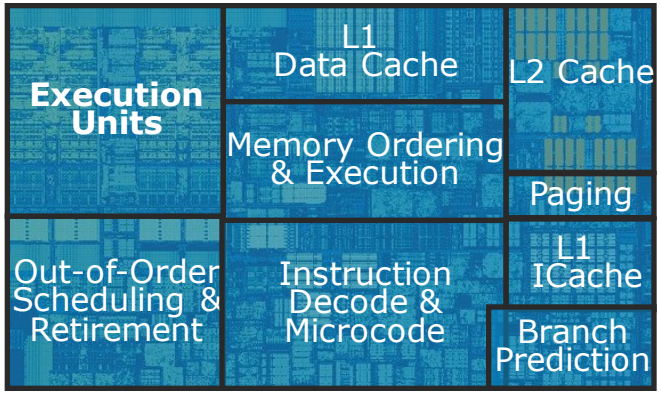


Putting It All Together: Intel Core i7 (Skylake)

- Each core has 19 pipeline stages, ~4GHz
- 6-wide superscalar
- Out of order execution
- Multi-level branch predictors
- Caches:
  - L1: 32KB I + 32KB D
  - L2: 256KB
  - L3: 8MB, shared
- Large overheads vs simple cores!



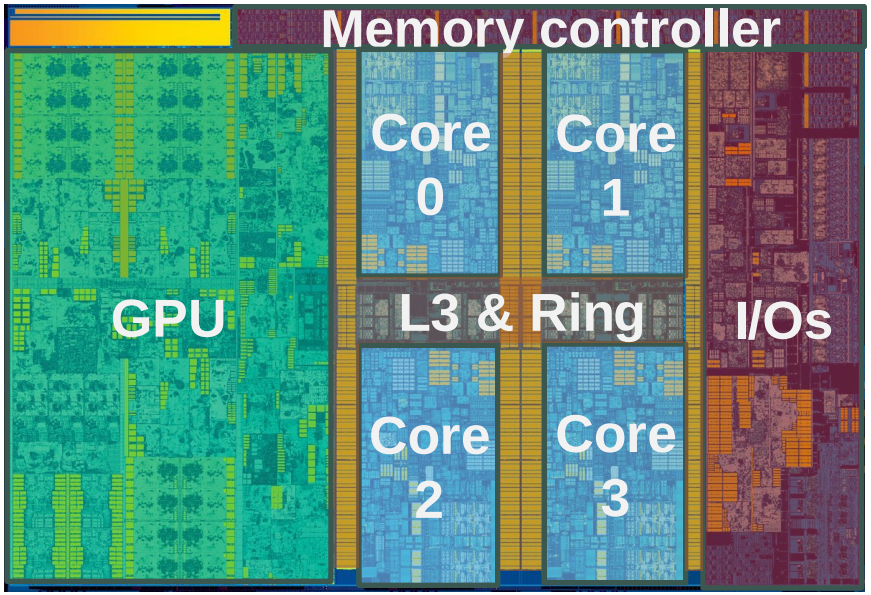
Intel, 2016, 14nm,
1.7B transistors, 122mm^2



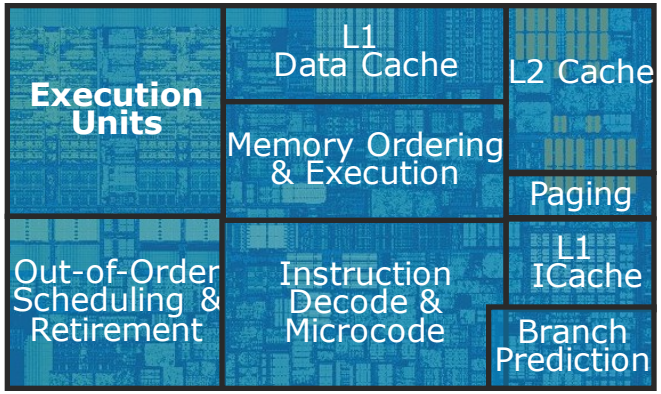
Putting It All Together: Intel Core i7 (Skylake)

- Each core has 19 pipeline stages, ~4GHz
- 6-wide superscalar
- Out of order execution
- Multi-level branch predictors
- Caches:
  - L1: 32KB I + 32KB D
  - L2: 256KB
  - L3: 8MB, shared

- Large overheads vs simple cores!



Intel, 2016, 14nm, 1.7B transistors, 122mm^2



- Your RISC-V core

Thank you!

Good luck on Quiz 3 ☺