

Pipeline Hazards

From Chapter 14, Computer Organization and
Architecture by William Stallings

Recap

- A fully utilized pipeline with 6 stages

Time →

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO

Figure 14.10 Timing Diagram for Instruction Pipeline Operation

Pipeline Hazard in Computer

- A pipeline hazard occurs when the pipeline, or some portion of the pipeline, must **stall** because conditions do not permit continued execution.
- This scenario is also referred to as a pipeline bubble.
- Three Hazard types
 - **Structural/Resource:** same resource is needed multiple times in the same cycle
 - **Data:** Instruction depends on result of prior instruction still in the pipeline. Data dependencies limit pipelining
 - **Control:** Next pipelined instruction may not be the next specified instruction

Pipeline Hazard in Computer

Can always resolve hazards by stalling

BUT

more stall cycles = more CPU time wasted = less
performance

To increase performance, try to decrease stall cycles

Resource Hazard: Real World Analogy

- Structural/Resource Hazard can be explained using car manufacture pipeline example.
- Assume that we have one employee that assembles and paints the car as well. We have a conflict of resource, P and A cannot be done at same time.
- One solution is that car 2 waits for A until P of car 1 is complete. Car 3 waits for A till P of Car 2 is complete
- Another solution can be to add more resources, in this case hire another employee for paint job.

Time	1 st hr	2 nd	3 rd	4 th	5 th
Car 1	A	P	T		
Car 2		A	P	T	
Car 3			A	P	T

Resource conflict

Time	1 st hr	2 nd	3 rd	4 th	5 th	6 th	7 th
Car 1	A	P	T				
Car 2		idle	A	P	T		
Car 3				idle	A	P	T

A possible solution

Structural/Resource Hazard

- A resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource.
- The result is that the instructions must be executed in serial rather than parallel for a portion of the pipeline.
- Examples:
 - Two accesses to a single ported memory
 - Two operations need the same function unit at the same time
 - Two operations need the same function unit in successive cycles, but the unit is not pipelined

Resource Hazard Example

	Clock cycle								
	1	2	3	4	5	6	7	8	9
	I1	FI	DI	FO	EI	WO			
	I2		FI	DI	FO	EI	WO		
	I3			FI	DI	FO	EI	WO	
	I4				FI	DI	FO	EI	WO

	Clock cycle								
	1	2	3	4	5	6	7	8	9
	I1	FI	DI	FO	EI	WO			
	I2		FI	DI	FO	EI	WO		
	I3			Idle	FI	DI	FO	EI	WO
	I4				FI	DI	FO	EI	WO

- Assume there is a single port to memory
- FO of I1 and FI of I3, both require reading from memory at cycle 3
- Assume that operands for rest of the instructions are registers.
- There is resource conflict between I1 and I3
- I3 must wait

Data Hazards

- Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on a non-pipelined processor.
- Three types of data dependencies defined in terms of how succeeding instruction depends on preceding instruction
 - **RAW**: Read after Write or Flow dependency
 - **WAR**: Write after Read or Anti dependency
 - **WAW**: Write after Write

Read After Write (RAW)

- Instr-J tries to read operand before Instr-H writes it

H: ADD **AX**, BX

J: SUB CX, **AX**

		Clock cycle									
		1	2	3	4	5	6	7	8	9	10
ADD EAX, EBX		FI	DI	FO	EI	WO					
SUB ECX, EAX			FI	DI	Idle		FO	EI	WO		
I3				FI			DI	FO	EI	WO	
I4							FI	DI	FO	EI	WO

Figure 14.16 Example of Data Hazard

Write After Read (WAR)

- An instruction reads a register or memory location and a succeeding instruction writes to that location.
- A hazard occurs if the write operation completes before the read operation takes place.
 - H: read from location X
 - J: write to location X
 - Incorrect answer if J finishes before H

Only possible in CPU designs that do **not** follow in-order processing pipeline

Write After Write (WAW)

- Two instructions both write to the same location.
- A hazard occurs if the write operations take place in the reverse order of the intended sequence.

H: write to location X

J: write to location X

- Incorrect answer if J finishes before H

Only possible in CPU designs that do **not** follow in-order processing pipeline

RAW Data Hazard Solutions

- Stalling
- Consider example
- 6 stage pipeline

[illegible]

RAW Data Hazard Solutions

- Stalling
- Consider example
- 6 stage pipeline

[illegible]

RAW Data Hazard Solutions

- Stalling
- Consider example
- 6 stage pipeline

[illegible]

RAW Data Hazard Solutions

- Stalling
- Consider example
- 6 stage pipeline

[illegible]

RAW Data Hazard Solutions

- Stalling
- Consider example
- 6 stage pipeline

Clock Cycle →	1	2	3	4	5	6	7	8	9	10
mov byte [var1], 7	FI	DI	CO	FO	EI	WO				
cmp byte [var1], 0		FI	DI	CO	stall	stall	FO	EI	WO	
jge label2			FI	DI						

RAW Data Hazard Solutions

- Stalling
- Consider example
- 6 stage pipeline

Clock Cycle →	1	2	3	4	5	6	7	8	9	10
mov byte [var1], 7	FI	DI	CO	FO	EI	WO				
cmp byte [var1], 0		FI	DI	CO	stall	stall	FO	EI	WO	
jge label2			FI	DI			CO	FO	EI	WO

RAW Data Hazard Solutions

- Stalling

Practice examples

- Solve data hazards in a 6 stage pipeline by stalling

(i)

`shr dx, 3`

`or dx, 7`

(ii)

`lodsw`

`mov ax, [bx]`

RAW Data Hazard Solutions

- Stalling

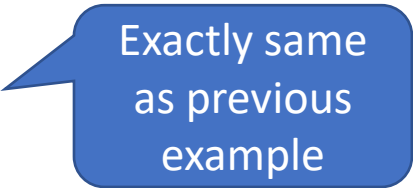
Practice examples

- Solve data hazards in a 6 stage pipeline by stalling

(i)

`shr dx, 3`

`or dx, 7`



Exactly same
as previous
example

(ii)

`lodsw`

`mov ax, [bx]`

RAW Data Hazard Solutions

- Stalling

Practice examples

- Solve data hazards in a 6 stage pipeline by stalling

(i)

shr dx, 3
or dx, 7

Exactly same
as previous
example

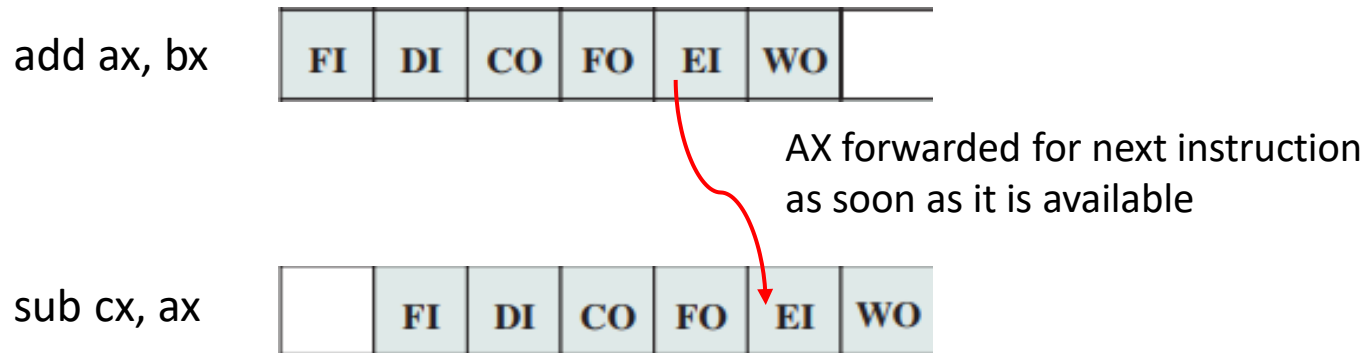
(ii)

lodsw
mov ax, [bx]

No RAW
dependency
-> no hazard

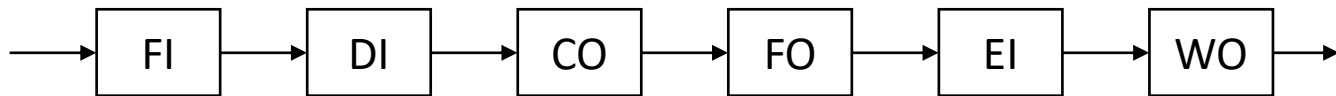
RAW Data Hazard Solutions

- Data Forwarding
 - Data is forwarded to next instruction as soon as its available so that next instruction doesn't have to wait for write operand



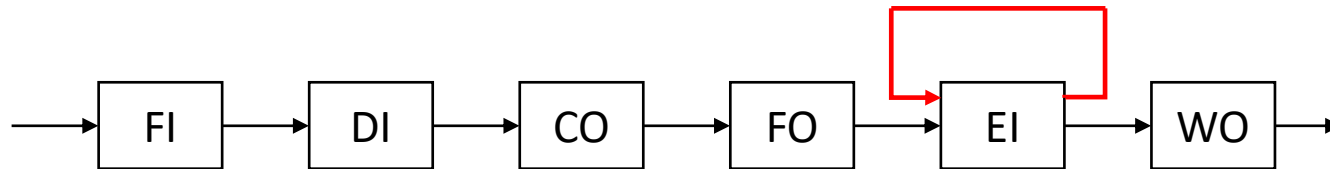
RAW Data Hazard Solutions

- Data Forwarding needs changes in hardware



RAW Data Hazard Solutions

- Data Forwarding needs changes in hardware
 - So that output of one stage is immediately available as input to same or a subsequent stage




one possible hardware change

RAW Data Hazard Solutions

- Data Forwarding example

Clock Cycle →	1	2	3	4	5	6	7	8	9	10
mov byte [var1], 7	FI	DI	CO	FO	EI	WO				
cmp byte [var1], 0		FI	DI	CO	FO	EI	WO			



RAW Data Hazard Solutions

Practice problem

- Solve data hazards in a 6 stage pipeline by data forwarding

shr dx, 4

or dx, 15

[illegible]

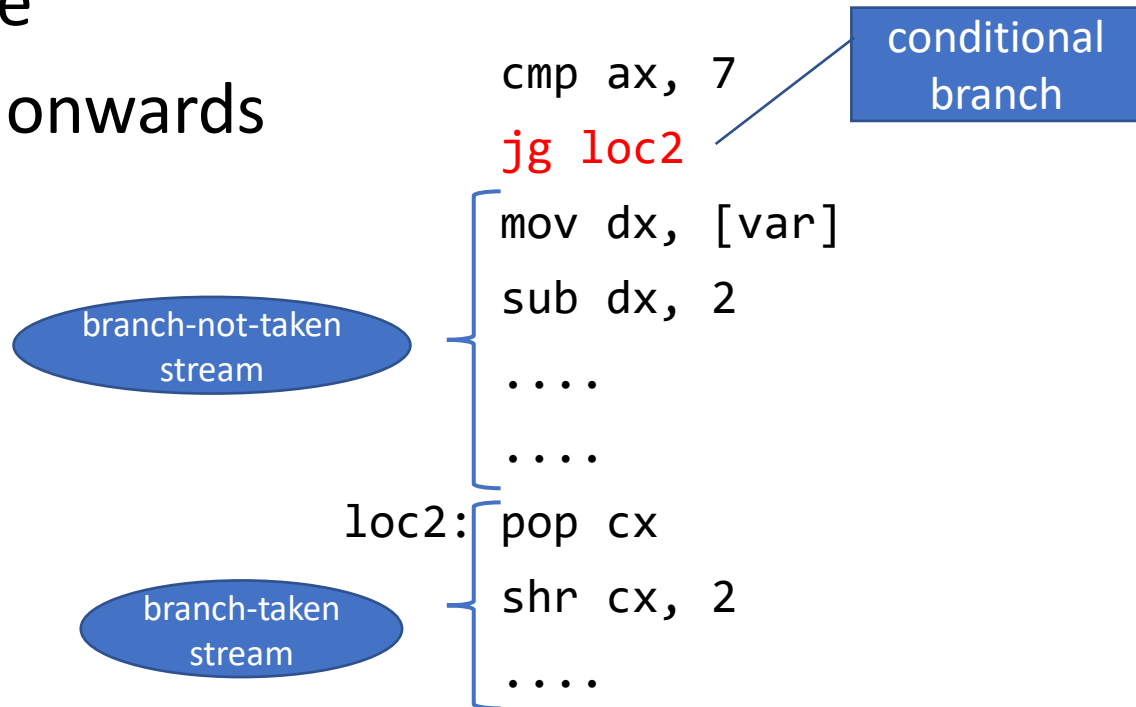
Control Hazard

- Control Hazards occur with conditional branch (jump) instructions like je, jne, jg, jle
- The pipeline must fetch the next instruction before it knows if jump will be taken or not.
- In case pipeline picks up the wrong stream, it brings instructions into the pipeline that must be discarded later on.
- Also known as a branch hazard

Control Hazard

CPU has to decide which of the two **instruction streams** to fetch in pipeline next:

1. Next in sequence
2. Branch target & onwards



Control Hazard: Example

	Time →						← Branch penalty							
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO							
Instruction 5					FI	DI	CO							
Instruction 6						FI	DI							
Instruction 7							FI							
Instruction 15								FI	DI	CO	FO	EI	WO	
Instruction 16									FI	DI	CO	FO	EI	WO

Figure 14.11 The Effect of a Conditional Branch on Instruction Pipeline Operation

- If instruction 3 was conditional branch
- Until the instruction is actually executed, it is impossible to determine whether the branch will be taken or not.
- If I3 takes jump to I15, all the work done for I4-I7 is wasted. Pipeline must be “flushed”

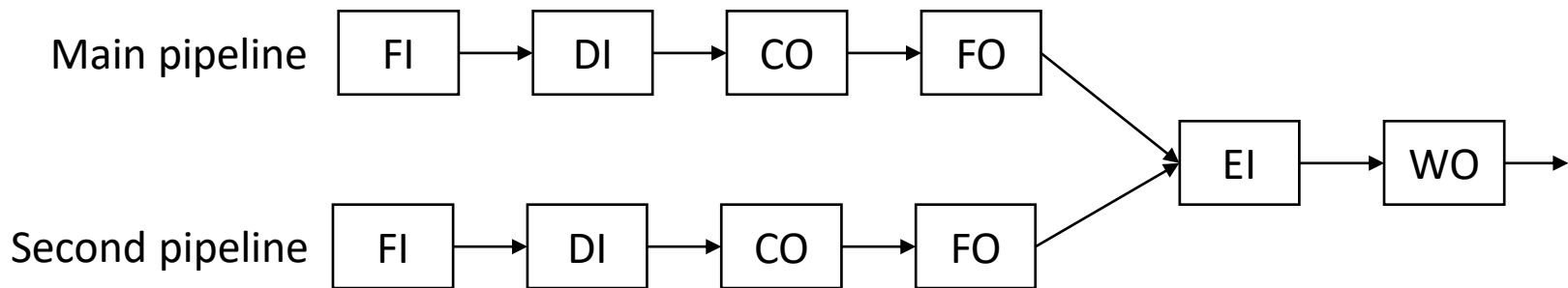
Branch Penalty: clock cycles lost due to incorrect branch prediction

Dealing with Branches

- A variety of approaches have been taken for dealing with conditional branches
 - Always stall (full penalty on every branch)
 - Multiple streams
 - Loop buffer
 - Branch prediction
- Remember, objective is to reduce the branch penalty

Multiple Streams

- Replicate initial pipeline stages (FI, DI, CO, FO).
- When branch is detected, next instructions are fetched from both possible locations (next in sequence and branch target).



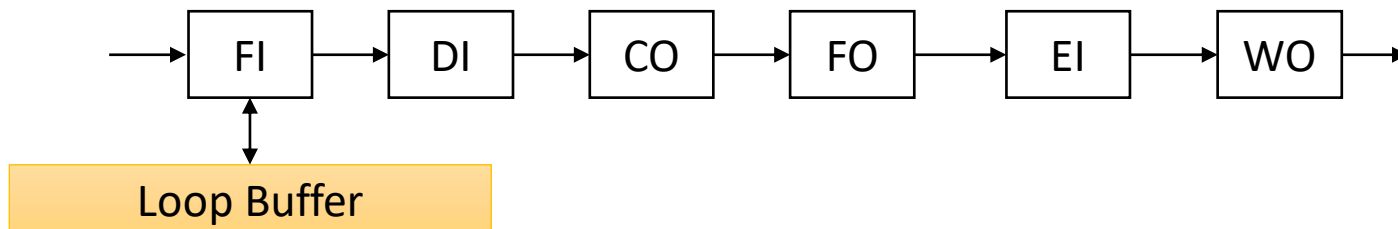
Multiple Streams

Problems:

- Contention of registers/memory etc. since duplicate stages need same hardware
- Additional branch instructions may enter pipeline before the original branch decision is resolved

Loop Buffer

- A small high speed memory maintained by the fetch instruction (FI) of the pipeline
- Contains n most recently fetched instructions
- If branch is to be taken, it is likely that branch target is within the loop buffer
 - If branch target is just a few locations ahead of current instruction, the target will already be in buffer. Useful for many IF-THEN or IF-ELSE structures
 - If buffer is large enough to contain a complete loop body, then instructions need to be fetched in first loop iteration only.



Branch Prediction

Predict whether the branch will be taken and fetch accordingly

- Static approaches
 - Predict never taken
 - Predict always taken
 - Predict by opcode
- Dynamic approaches
 - Taken/not taken switch

Static Branch Prediction

- Simplest strategies
 - Predict never taken – continue to fetch instructions in sequence
 - Predict always taken – start fetching instructions from branch target onwards
 - Studies have shown this prediction is true more than 50% of the time

Static Branch Prediction

- Example: Predict never-taken
- In case, branch was really not taken
- No penalty!

```
jg loc2
mov dx, [var]
sub dx, 2
...
...
pop cx
shr cx, 2
...
```

[illegible]

Static Branch Prediction

- Example: Predict never-taken
- In case, branch was actually taken
- **Penalty of 4 cycles!**

```

jg loc2
mov dx, [var]
sub dx, 2
....
....
loc2: pop cx
      shr cx, 2
      ....
    
```

Clock Cycle →	1	2	3	4	5	6	7	8	9	10
jg loc2	FI	DI	CO	FO	EI	WO				
mov dx, [var]		FI	DI	CO	FO					
sub dx, 2			FI	DI	CO					
...				FI	DI					
...					FI					
pop cx						FI	DI	CO	FO	EI
shr cx, 2							FI	DI	FO	

Static Branch Prediction

- Predict by opcode – assume that branch will be taken for certain opcodes and not others
 - Reported success rate of 75%

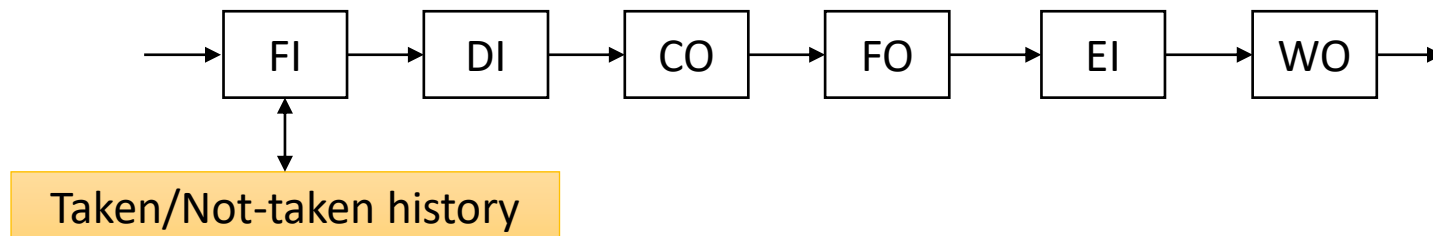
Dynamic Branch Prediction

Taken/not taken switch

- One or more bits associated with each conditional branch instruction, that tell the recent history of that instruction
- Branch prediction is based on that behavior

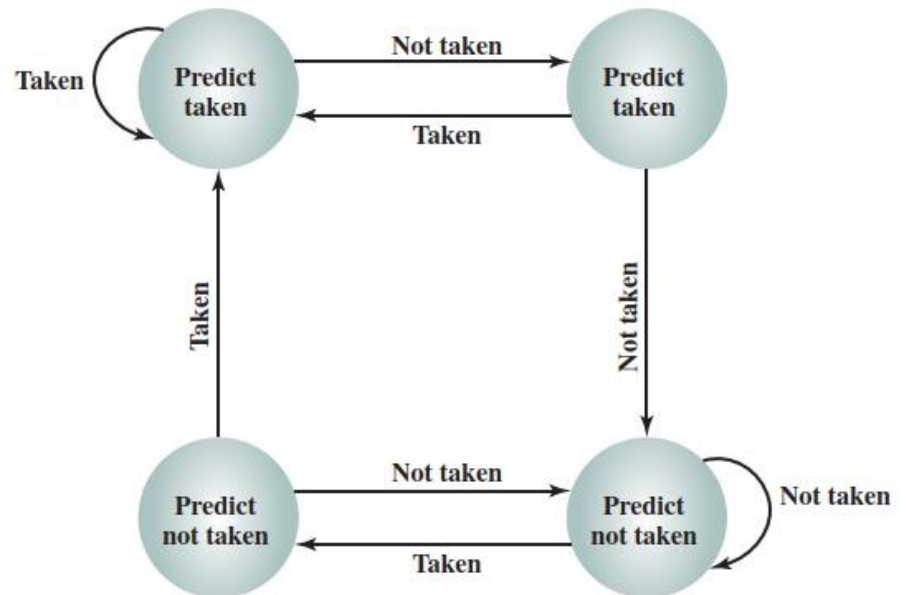
Instruction Address	Was branch taken?
0x252D3	1
0x252E0	0
0x252F5	0
.....	1

Taken/not taken switch
with 1 bit history



Dynamic Branch Prediction

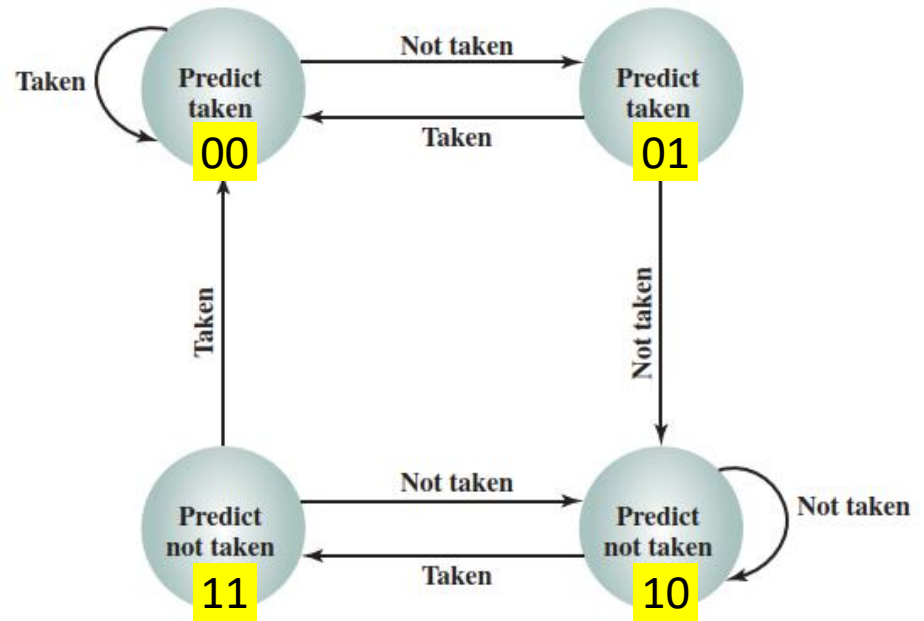
- Taken/not taken switch
- With two bits of history, we can record the result of the last two instances of execution of an instruction
- Algorithm requires two consecutive wrong predictions to change its prediction decision



History based prediction example

AX CX

➔ 100 mov ax, 0
103 mov cx, 5
106 L2: inc ax
107 mov bx, 'A'
10A cmp ax, 8
10D jg L3
10F mov bx, 'B'
112 L3: call printchar
115 loop L2



Instruction offset	State

Table initially empty

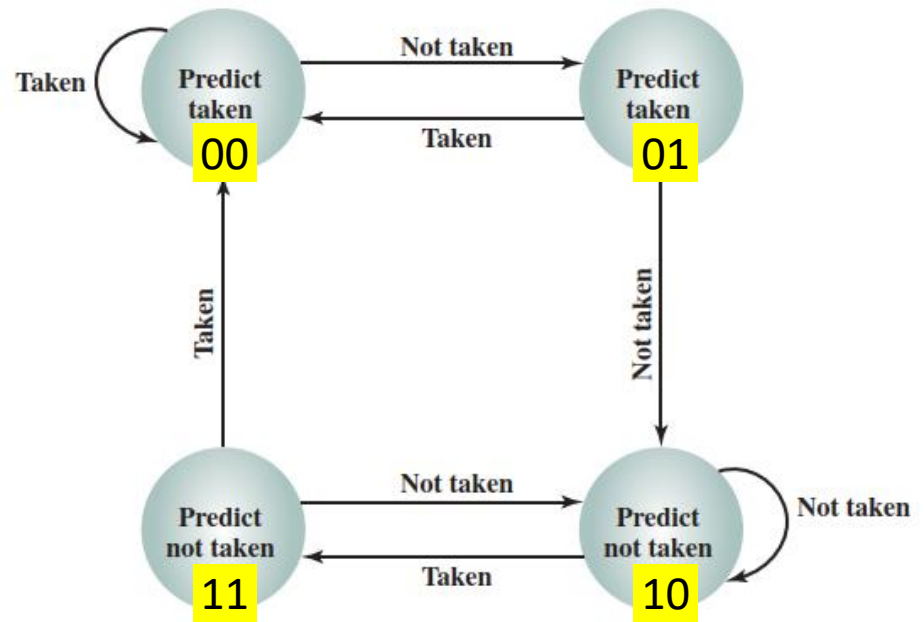
Branching History

History based prediction example

AX 0000 CX

➔

100	mov ax, 0
103	mov cx, 5
106	L2: inc ax
107	mov bx, 'A'
10A	cmp ax, 8
10D	jg L3
10F	mov bx, 'B'
112	L3: call printchar
115	loop L2



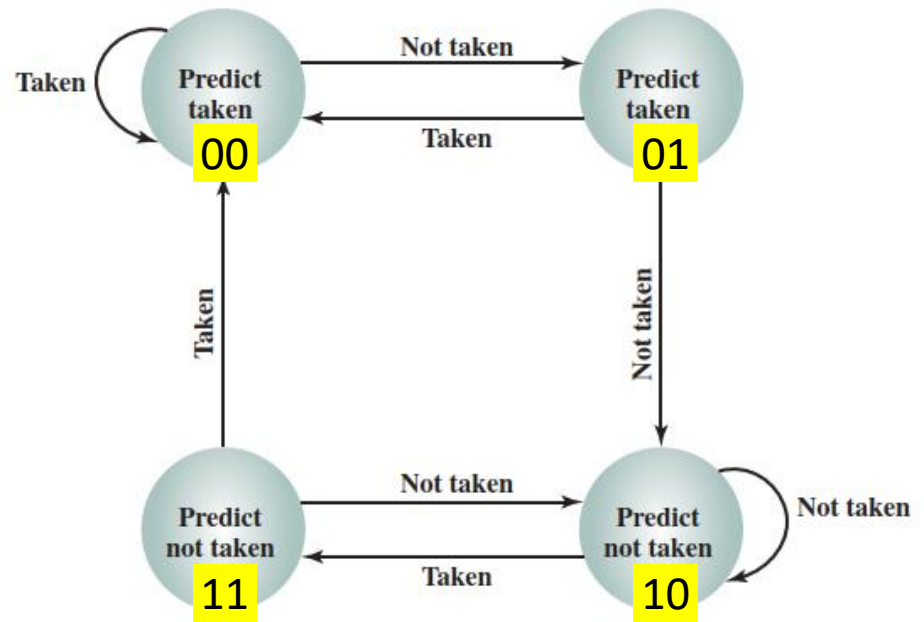
Instruction offset	State

Branching History

History based prediction example

AX 0001 CX 0005

100	mov ax, 0
103	mov cx, 5
106	L2: inc ax
107	mov bx, 'A'
10A	cmp ax, 8
→ 10D	jg L3
10F	mov bx, 'B'
112	L3: call printchar
115	loop L2



Instruction offset	State

Branching History

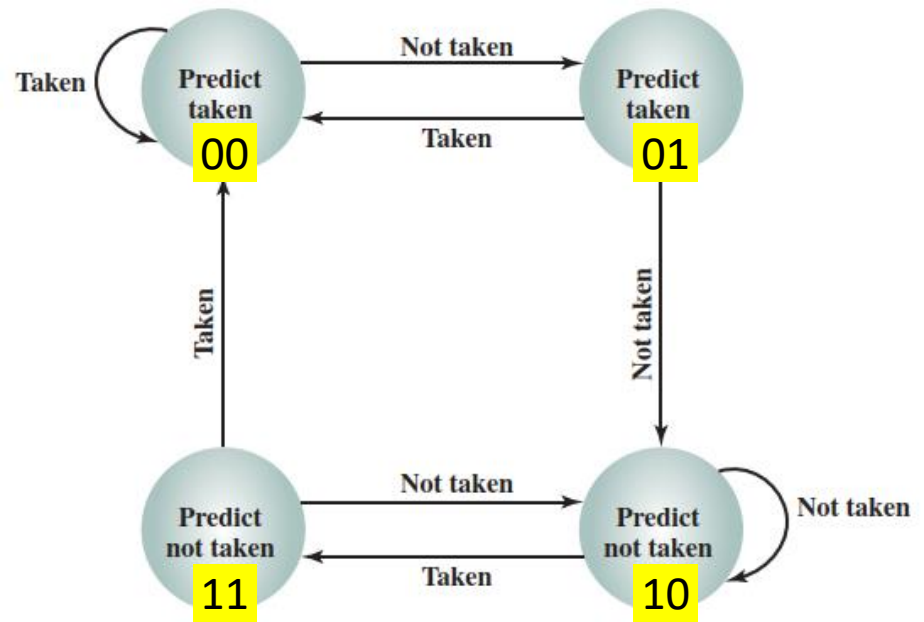
History based prediction example

AX	0001	CX	0005
----	------	----	------

```

100      mov ax, 0
103      mov cx, 5
106      L2: inc ax
107      mov bx, 'A'
10A      cmp ax, 8
10D      jg L3
10F      mov bx, 'B'
112      L3: call printchar
115      loop L2

```



Instruction offset	State
10D	00

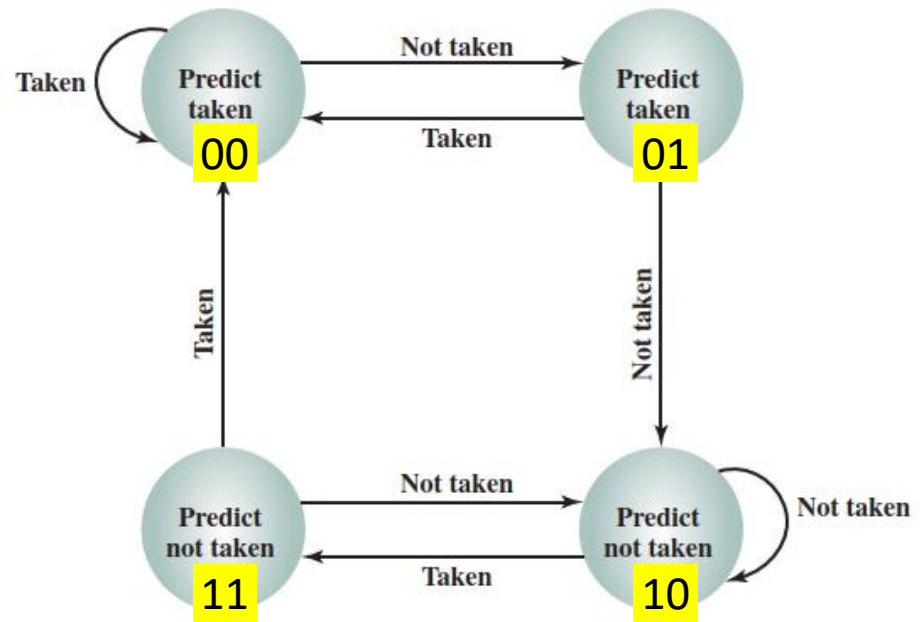
Default state:
predict taken

Branching History

History based prediction example

AX 0001 CX 0005

100	mov ax, 0
103	mov cx, 5
106	L2: inc ax
107	mov bx, 'A'
10A	cmp ax, 8
10D	jg L3
10F	mov bx, 'B'
112	L3: call printchar
115	loop L2



Instruction offset	State
10D	01

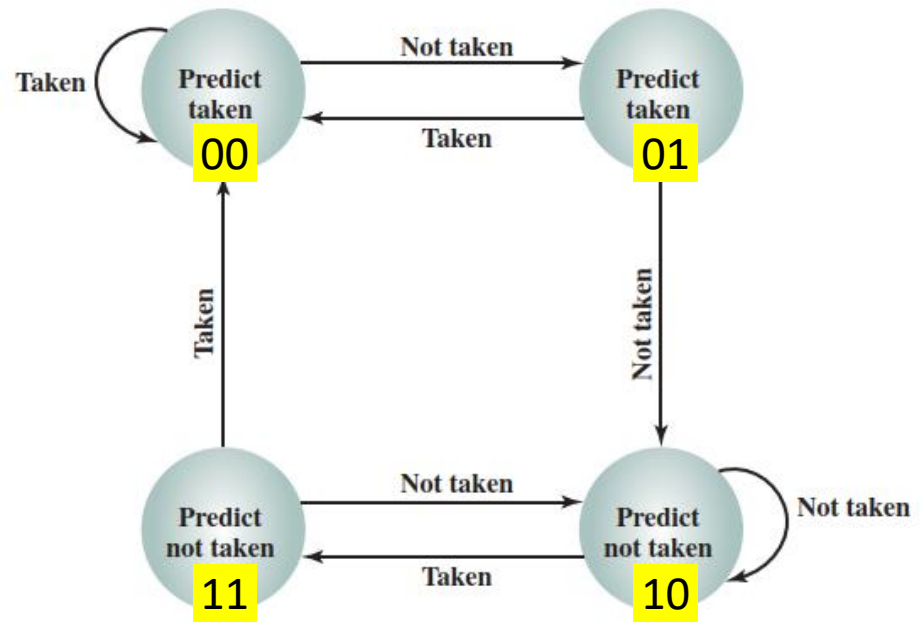
Actual outcome:
not taken.
State changed

Branching History

History based prediction example

AX 0001 CX 0005

100	mov ax, 0
103	mov cx, 5
106	L2: inc ax
107	mov bx, 'A'
10A	cmp ax, 8
10D	jg L3
10F	mov bx, 'B'
112	L3: call printchar
115	loop L2



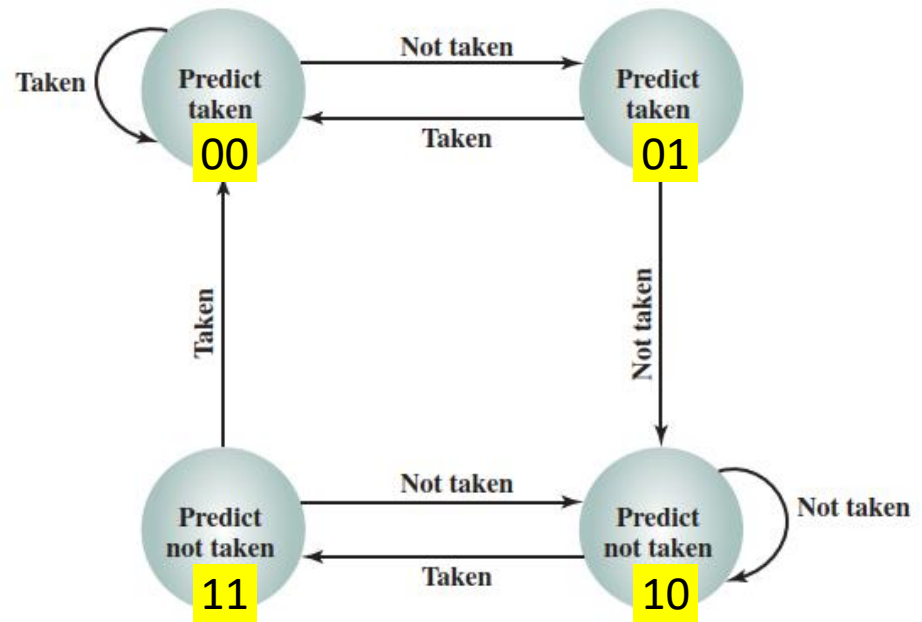
Instruction offset	State
10D	01

Branching History

History based prediction example

AX 0001 CX 0005

100	mov ax, 0
103	mov cx, 5
106	L2: inc ax
107	mov bx, 'A'
10A	cmp ax, 8
10D	jg L3
10F	mov bx, 'B'
112	L3: call printchar
115	loop L2



Instruction offset	State
10D	01
115	00

Default state:
predict taken

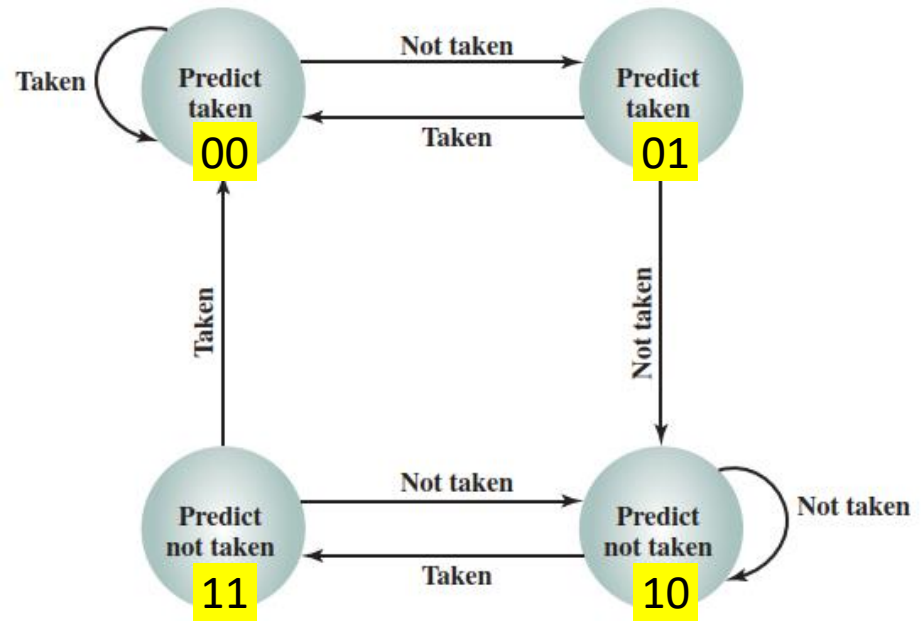
Branching History

History based prediction example

AX 0001 CX 0004

➔

100	mov ax, 0
103	mov cx, 5
106	L2: inc ax
107	mov bx, 'A'
10A	cmp ax, 8
10D	jg L3
10F	mov bx, 'B'
112	L3: call printchar
115	loop L2



Instruction offset	State
10D	01
115	00

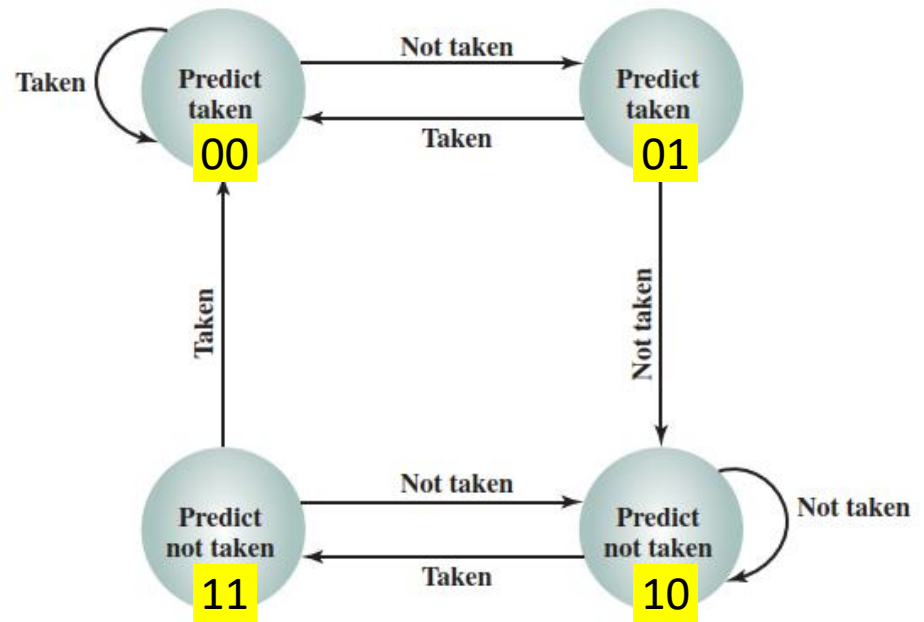
Actual outcome:
taken.
State unchanged

Branching History

History based prediction example

AX 0002 CX 0004

100 mov ax, 0
103 mov cx, 5
106 L2: inc ax
107 mov bx, 'A'
10A cmp ax, 8
10D jg L3
10F mov bx, 'B'
112 L3: call printchar
115 loop L2



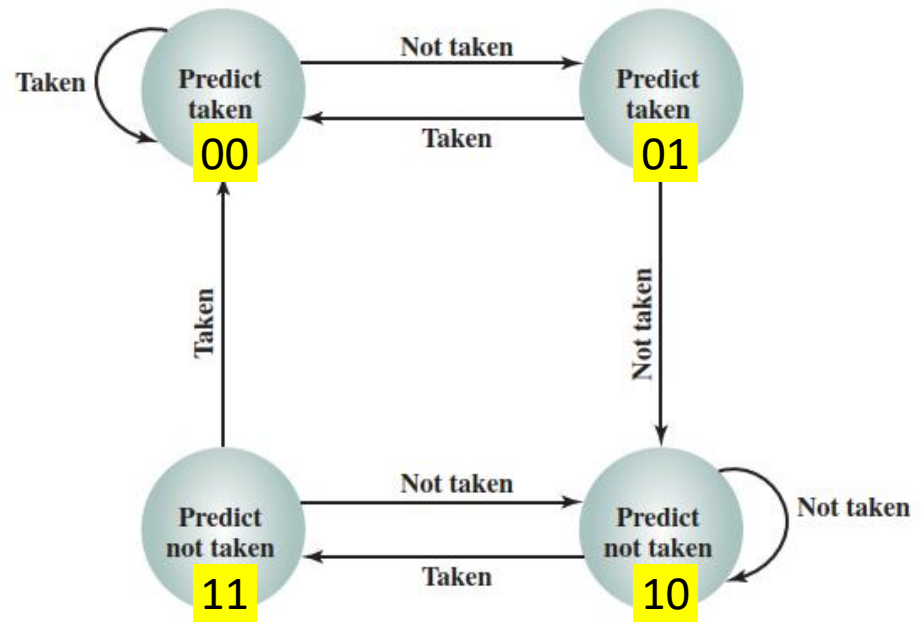
Instruction offset	State
10D	01
115	00

Branching History

History based prediction example

AX 0002 CX 0004

100		mov ax, 0
103		mov cx, 5
106		L2: inc ax
107		mov bx, 'A'
10A		cmp ax, 8
→ 10D		jg L3
10F		mov bx, 'B'
112		L3: call printchar
115		loop L2



Instruction offset	State
10D	01
115	00

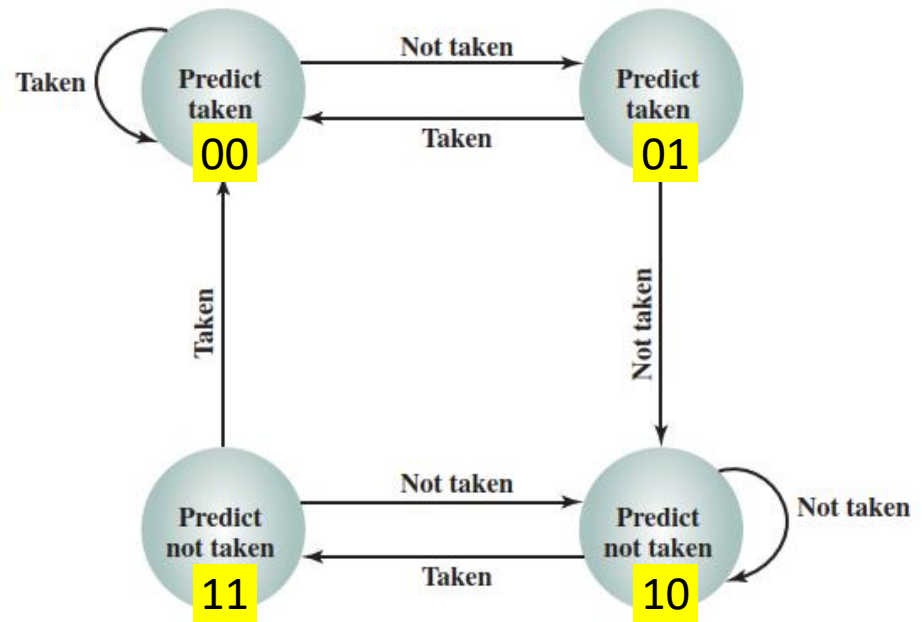
Branching History

History based prediction example

AX 0002 CX 0004

100	mov ax, 0
103	mov cx, 5
106	L2: inc ax
107	mov bx, 'A'
10A	cmp ax, 8
10D	jg L3
10F	mov bx, 'B'
112	L3: call printchar
115	loop L2

➔



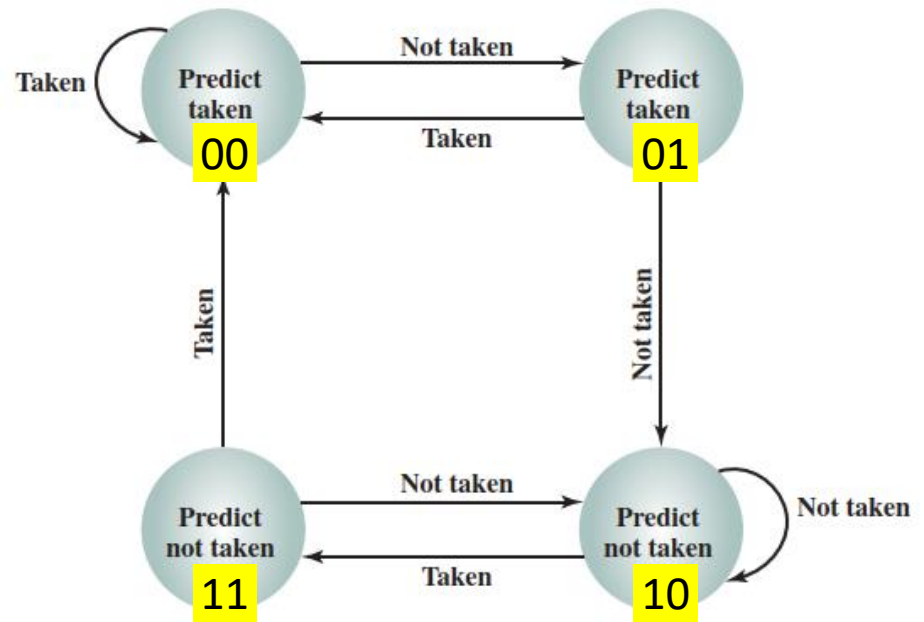
Instruction offset	State	
10D	01	predict taken
115	00	

Branching History

History based prediction example

AX 0002 CX 0004

100		mov ax, 0
103		mov cx, 5
106		L2: inc ax
107		mov bx, 'A'
10A		cmp ax, 8
10D		jg L3
→ 10F		mov bx, 'B'
112		L3: call printchar
115		loop L2



Instruction offset	State
10D	10
115	00

Actual outcome:
not taken.
State changed

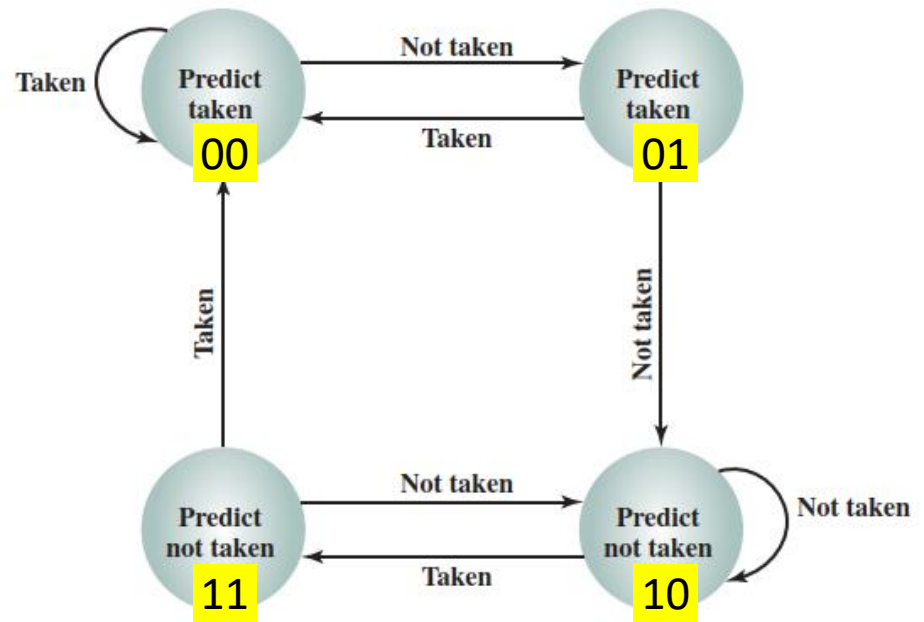
Branching History

History based prediction example

AX 0002 CX 0004

100		mov ax, 0
103		mov cx, 5
106		L2: inc ax
107		mov bx, 'A'
10A		cmp ax, 8
10D		jg L3
10F		mov bx, 'B'
112		L3: call printchar
115		loop L2

➔



Instruction offset	State
10D	10
115	00

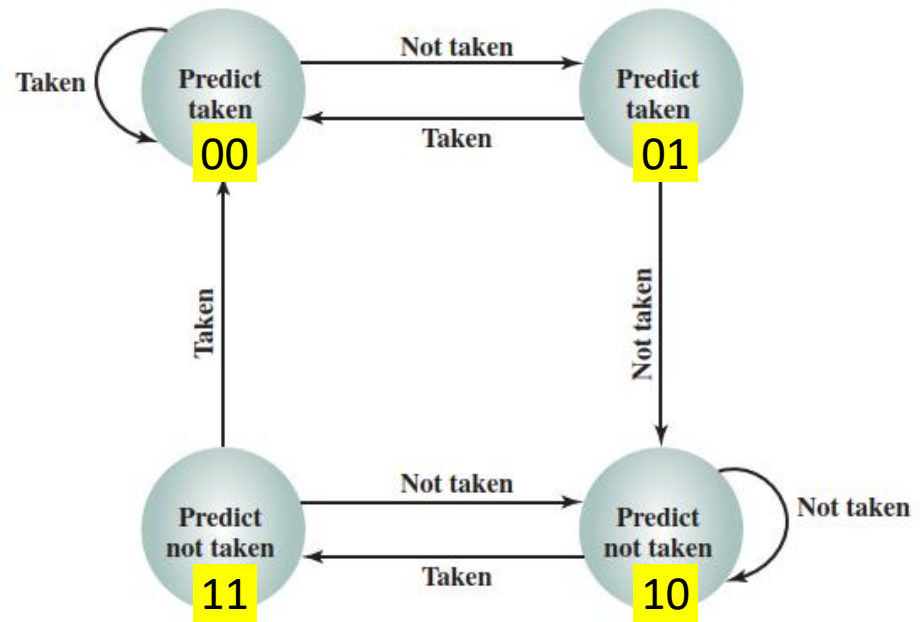
Branching History

History based prediction example

AX 0002 CX 0004

100		mov ax, 0
103		mov cx, 5
106		L2: inc ax
107		mov bx, 'A'
10A		cmp ax, 8
10D		jg L3
10F		mov bx, 'B'
112		L3: call printchar
115		loop L2

➔



Instruction offset	State	
10D	10	
115	00	predict taken

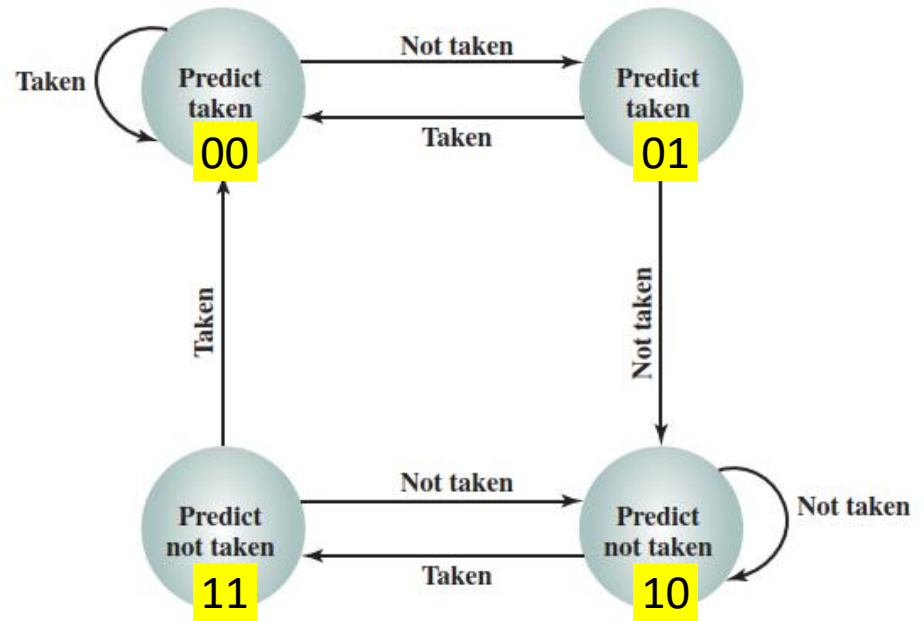
Branching History

History based prediction example

AX 0002 CX 0003

➔

100	mov ax, 0
103	mov cx, 5
106	L2: inc ax
107	mov bx, 'A'
10A	cmp ax, 8
10D	jg L3
10F	mov bx, 'B'
112	L3: call printchar
115	loop L2



Instruction offset	State
10D	10
115	00

Actual outcome:
taken.
State unchanged

Branching History

Reference

- Section 14.4 (Pipeline Hazards) from Computer Organization and Architecture Designing for Performance Tenth Edition, by William Stalling
- http://www.cs.uni.edu/~fienup/cs240s05/lectures/lec5_1-25-05_web.htm
- https://en.wikipedia.org/wiki/Branch_predictor