Enhancing Performance with Pipelining

Architettura dei Calcolatori Elettronici

Università
degli Studi
della Campania
Luigi Vanvitelli

Hazards

- …when the next instruction can not execute in the following clock cycle
  - control hazards:
    - *it arises from the need to make a decision based on the result of one instruction while others are executing*
    - e.g. what about branches?
  - structural hazards
    - *the hw can not support the combination of instructions that we want to execute in the same clock cycle*
    - e.g. what if we had only one memory?
  - data hazards:
    - *an instruction depends on the result of a previous instruction still in the pipeline*
    - e.g. an instruction's input operand is the output of a previous instruction

Hazards

- Can always resolve hazards by waiting
  - pipeline control must detect the hazard
  - and take action to resolve hazards

Data Hazards

Architettura dei Calcolatori Elettronici

Università
degli Studi
della Campania
Luigi Vanvitelli

Hazards

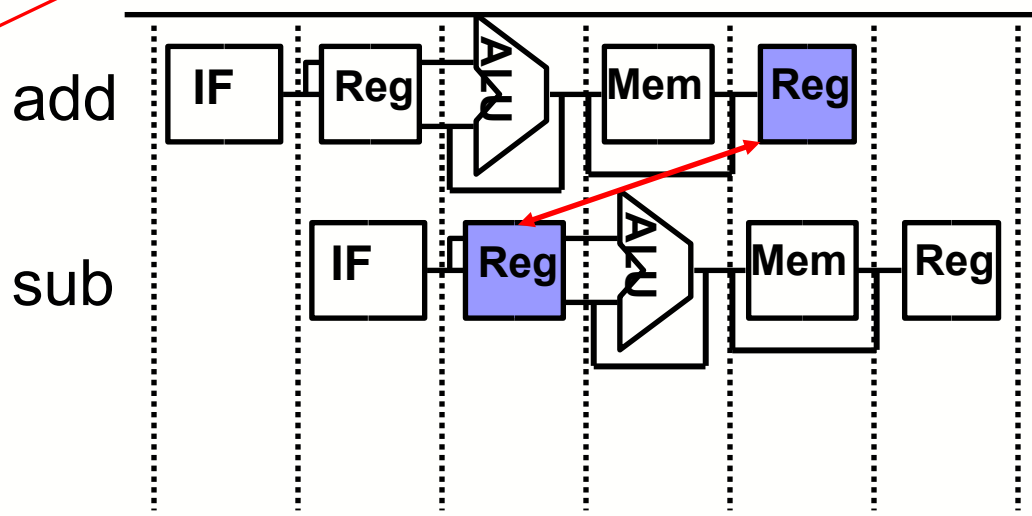
- …when the next instruction can not execute in the following clock cycle
  - control hazards:
    - *it arises from the need to make a decision based on the result of one instruction while others are executing*
    - e.g. what about branches?
  - structural hazards
    - *the hw can not support the combination of instructions that we want to execute in the same clock cycle*
    - e.g. what if we had only one memory?
  - data hazards:
    - *an instruction depends on the result of a previous instruction still in the pipeline*
    - e.g. an instruction's input operand is the output of a previous instruction

Example

Read after Write Data Hazard

add $s0, $t0, $t1

sub $t2, $s0, $t3



- new value of $s0 is unavailable before clock cycle 5
- sub uses the new value of $s0 in clock cycle 2
- **sub** is dependent on the result of the **add**

Solutions

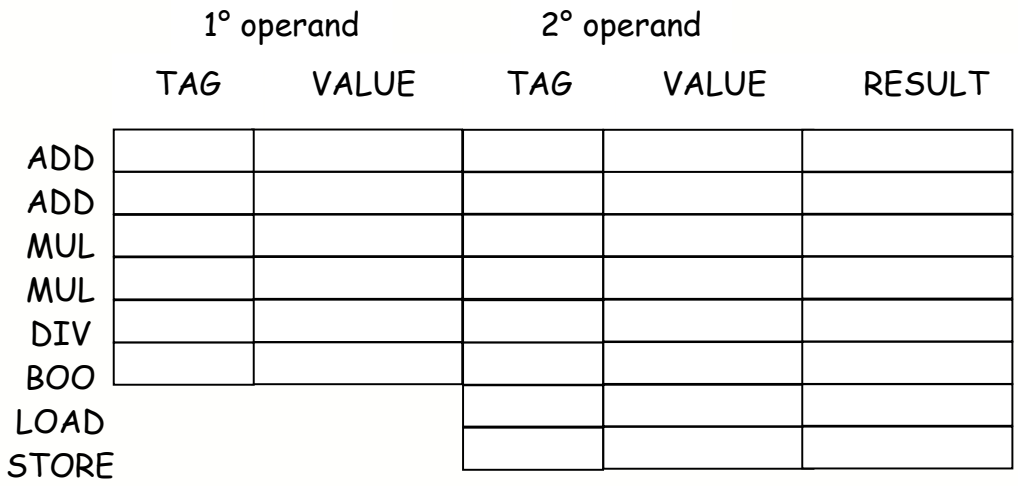
- Operand Forwarding (already addressed in the previous lesson)
  - or Bypassing or ShortCircuiting
  - Hardware modifications

- Internal Forwarding
  - Hibernation of not executable instructions
  - A table and additional registers

Internal Forwarding

- Hibernation table
- Forwarding registers
- Operand registers

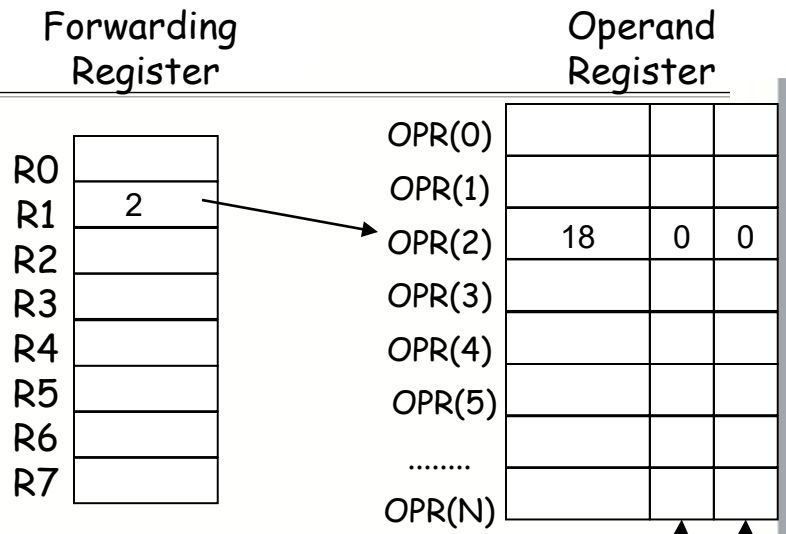
Hibernation Table



- We can hibernate:
  - 1 load operation
  - 1 store operation
  - 1 boolean operation
  - 1 division operation
  - 2 multiplication operation
  - 2 addition operations
- Only 1 load/store operation
  - until a memory access is not concluded, it is impossible to begin another memory access

Registers

- Forwarding Registers
  - They are used as pointers to Operand Registers
- Operand Registers
  - They contain values
    - current values and past values
  - Number of suspended operations
  - Valid bit
    - 0 valid
    - 1 not valid

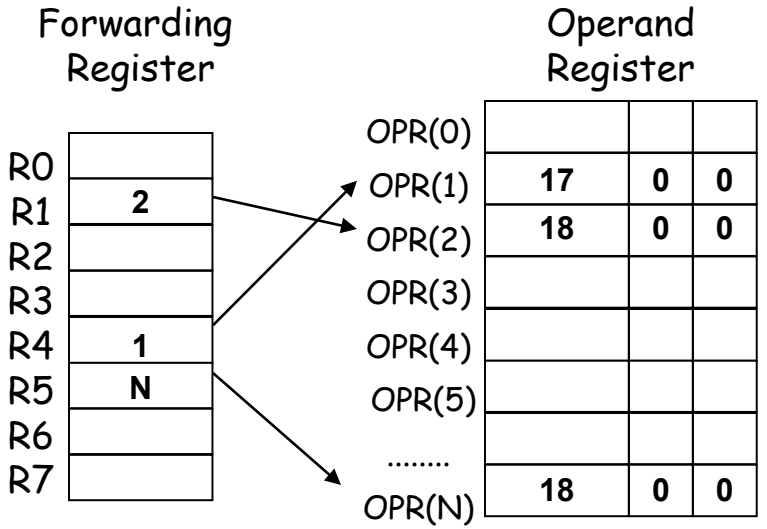


Example

```
lw    $r1, MemA
mul   $r2, $r1, $r5
div   $r3, $r2, $r5
add   $r4, $r3, $r4
```

	1° operand		2° operand		
	TAG	VALUE	TAG	VALUE	RESULT
ADD					
ADD					
MUL					
MUL					
DIV					
BOO					
LOAD					
STORE					

- old value of r1 = 18
- old value of r4 = 17



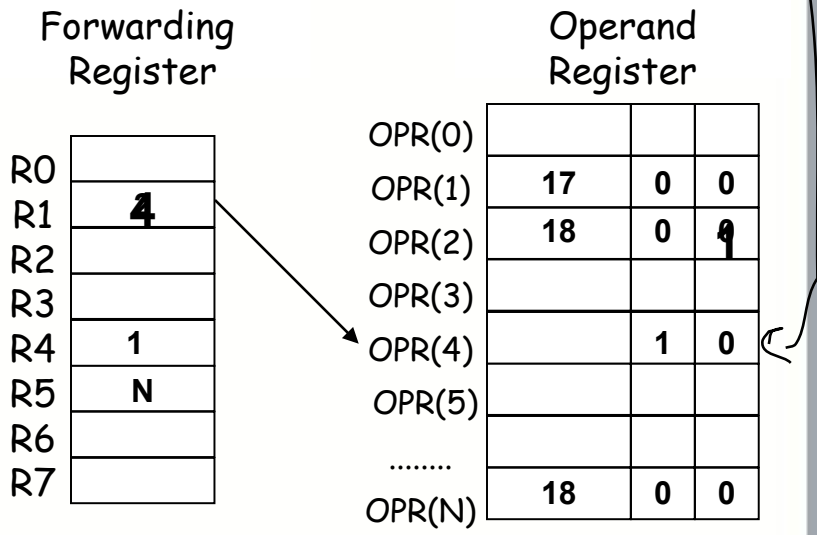
Example

cache miss

```
lw    $r1, MemA
mul   $r2, $r1, $r5
div   $r3, $r2, $r5
add   $r4, $r3, $r4
```

	1° operand		2° operand		
	TAG	VALUE	TAG	VALUE	RESULT
ADD					
ADD					
MUL					
MUL					
DIV					
BOO					
LOAD			MemA		4
STORE					

- Cache miss
- Load is hibernated
- Result will be stored in OPR(4)

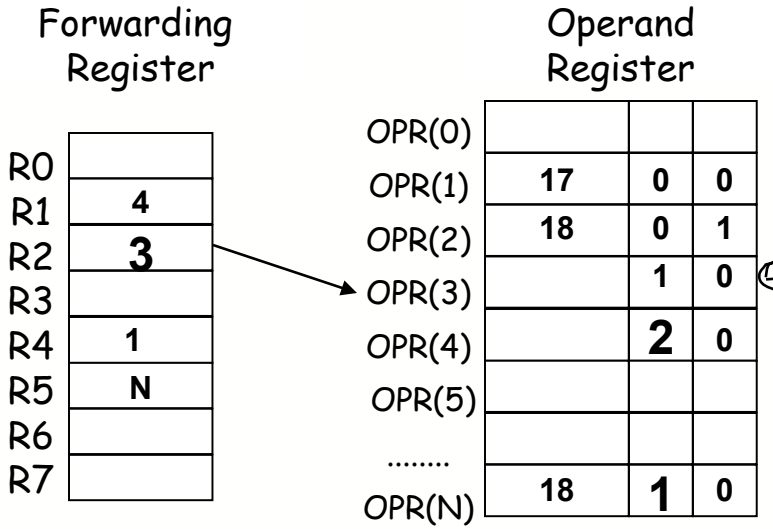


Example

```
lw      $r1, MemA
mul     $r2, $r1, $r5
div     $r3, $r2, $r5
add     $r4, $r3, $r4
```

	1° operand TAG	1° operand VALUE	2° operand TAG	2° operand VALUE	RESULT
ADD					
ADD					
MUL	4		N	18	3
MUL					
DIV					
BOO					
LOAD			MemA		4
STORE					

- mul is hibernated
- Result will be stored in OPR(3)

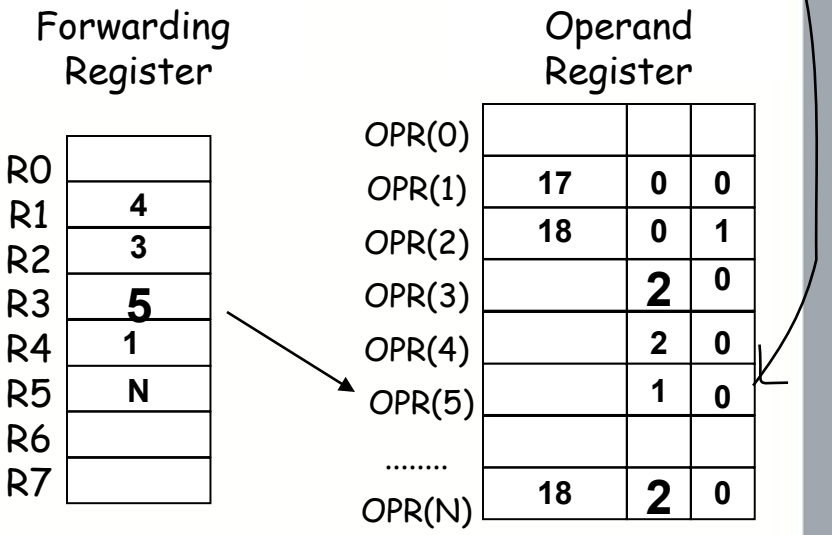


Example

```
lw    $r1, MemA
mul   $r2, $r1, $r5
div   $r3, $r2, $r5
add   $r4, $r3, $r4
```

	1° operand TAG	1° operand VALUE	2° operand TAG	2° operand VALUE	RESULT
ADD					
ADD					
MUL	4		N	18	3
MUL					
DIV	3		N	18	5
BOO					
LOAD			MemA		4
STORE					

- div is hibernated
- Result will be stored in OPR(5)

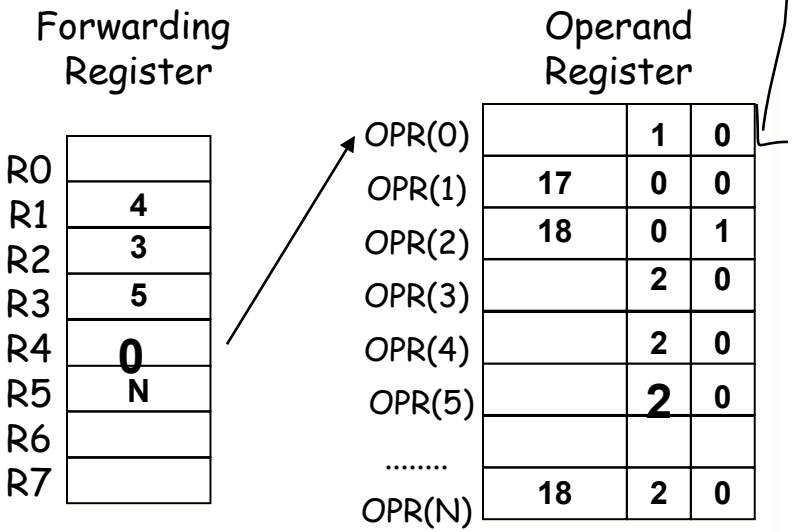


Example

```
lw    $r1, MemA
mul   $r2, $r1, $r5
div   $r3, $r2, $r5
add   $r4, $r3, $r4
```

	1° operand TAG	1° operand VALUE	2° operand TAG	2° operand VALUE	RESULT
ADD	5		1	17	0
ADD					
MUL	4		N	18	3
MUL					
DIV	3		N	18	5
BOO					
LOAD			MemA		4
STORE					

- add is hibernated
- Result will be stored in OPR(0)



Example

```
lw    $r1, MemA
mul   $r2, $r1, $r5
div   $r3, $r2, $r5
add   $r4, $r3, $r4
```

	1° operand		2° operand		
	TAG	VALUE	TAG	VALUE	RESULT
ADD	5		1	17	0
ADD					
MUL	4		N	18	3
MUL					
DIV	3		N	18	5
BOO					
LOAD					
STORE					

- MemA available

Forwarding Register

R0	
R1	4
R2	3
R3	5
R4	0
R5	N
R6	
R7	

Operand Register

OPR(0)		1	0
OPR(1)	17	0	0
OPR(2)	18	0	1
OPR(3)		2	0
OPR(4)	20	1	0
OPR(5)		2	0
.....			
OPR(N)	18	2	0

Example

```
lw      $r1, MemA
mul     $r2, $r1, $r5
div     $r3, $r2, $r5
add     $r4, $r3, $r4
```

	1° operand		2° operand		
	TAG	VALUE	TAG	VALUE	RESULT
ADD	5		1	17	0
ADD					
MUL					
MUL					
DIV	3		N	18	5
BOO					
LOAD					
STORE					

Forwarding Register

R0	
R1	4
R2	3
R3	5
R4	0
R5	N
R6	
R7	

Operand Register

OPR(0)		1	0
OPR(1)	17	0	0
OPR(2)	18	0	1
OPR(3)	360	1	0
OPR(4)	20	0	0
OPR(5)		2	0
.....			
OPR(N)	18	1	0

Example

```
lw      $r1, MemA
mul     $r2, $r1, $r5
div     $r3, $r2, $r5
add     $r4, $r3, $r4
```

	1° operand		2° operand		
	TAG	VALUE	TAG	VALUE	RESULT
ADD	5		1	17	0
ADD					
MUL					
MUL					
DIV					
BOO					
LOAD					
STORE					

Forwarding Register

R0	
R1	4
R2	3
R3	5
R4	0
R5	N
R6	
R7	

Operand Register

OPR(0)		1	0
OPR(1)	17	0	0
OPR(2)	18	0	1
OPR(3)	360	0	0
OPR(4)	20	0	0
OPR(5)	20	1	0
.....			
OPR(N)	18	0	0

Example

```
lw    $r1, MemA
mul   $r2, $r1, $r5
div   $r3, $r2, $r5
add   $r4, $r3, $r4
```

	1° operand		2° operand		
	TAG	VALUE	TAG	VALUE	RESULT
ADD					
ADD					
MUL					
MUL					
DIV					
BOO					
LOAD					
STORE					

Forwarding Register

R0	
R1	4
R2	3
R3	5
R4	0
R5	N
R6	
R7	

Operand Register

OPR(0)	37	0	0
OPR(1)	17	0	0
OPR(2)	18	0	1
OPR(3)	360	0	0
OPR(4)	20	0	0
OPR(5)	20	0	0
.....			
OPR(N)	18	0	0

All Data Hazards

- Read After Write
  - previous example!
- Read After Read
  - No hazards!
  - Read operation doesn’t change memory / registers status
- Write After Write
  - No hazards!
  - 2° Write operation is important.
- Write After Read

Control Hazards

Architettura dei Calcolatori Elettronici

Università
degli Studi
della Campania
Luigi Vanvitelli

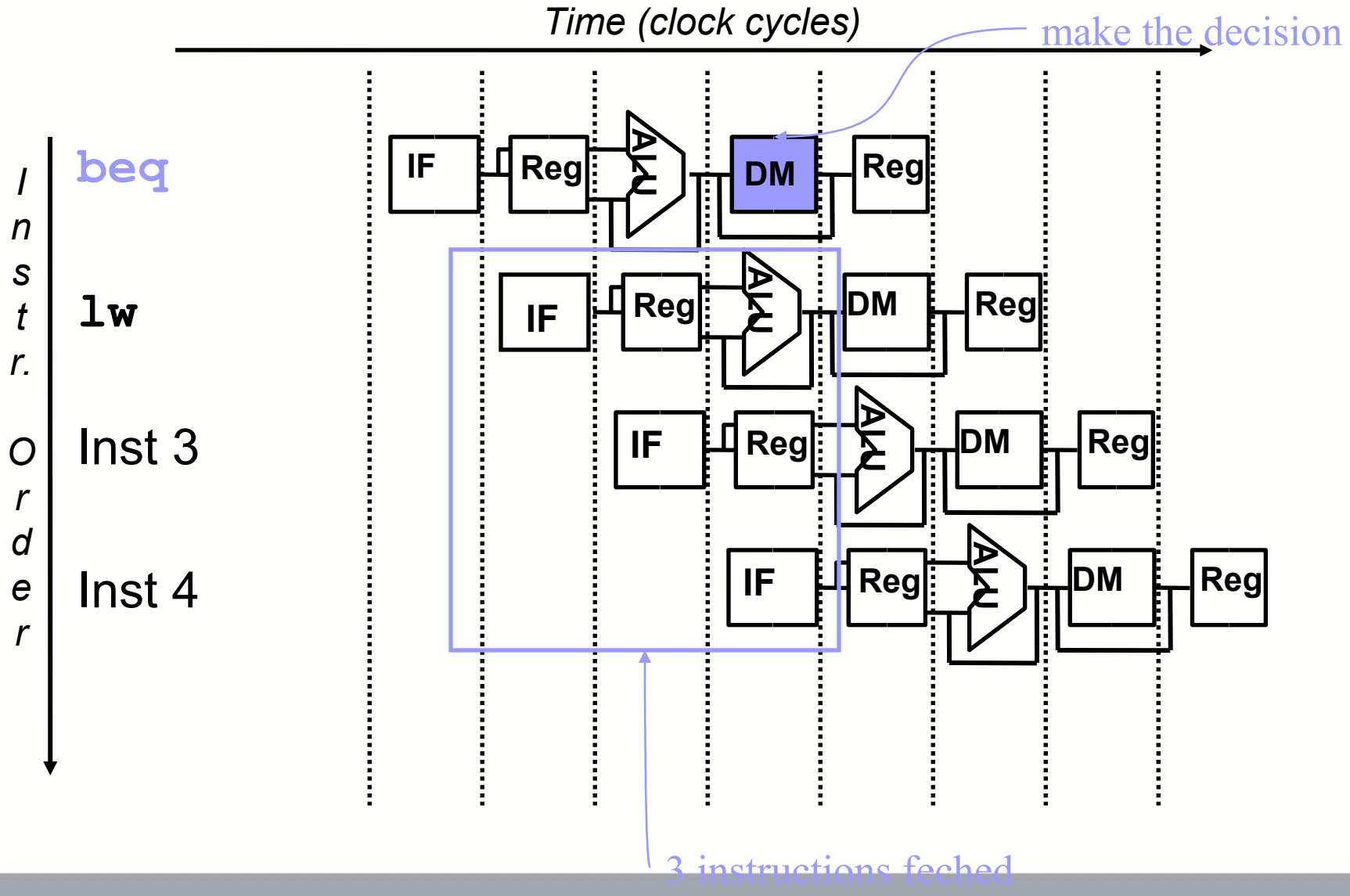
Hazards

- …when the next instruction can not execute in the following clock cycle
  - control hazards:
    - *it arises from the need to make a decision based on the result of one instruction while others are executing*
    - e.g. what about branches?
  - structural hazards
    - *the hw can not support the combination of instructions that we want to execute in the same clock cycle*
    - e.g. what if we had only one memory?
  - data hazards:
    - *an instruction depends on the result of a previous instruction still in the pipeline*
    - e.g. an instruction's input operand is the output of a previous instruction

Control Hazards

- When the flow of instruction addresses is not sequential
  - Conditional branches (beq, bne)
  - Unconditional branches (j, jal, jr)
  - Exceptions
- An instruction must be fetched at every clock cycle to sustain the pipeline
- The decision about whether to branch doesn't occur until the 4° stage

Branch Instructions



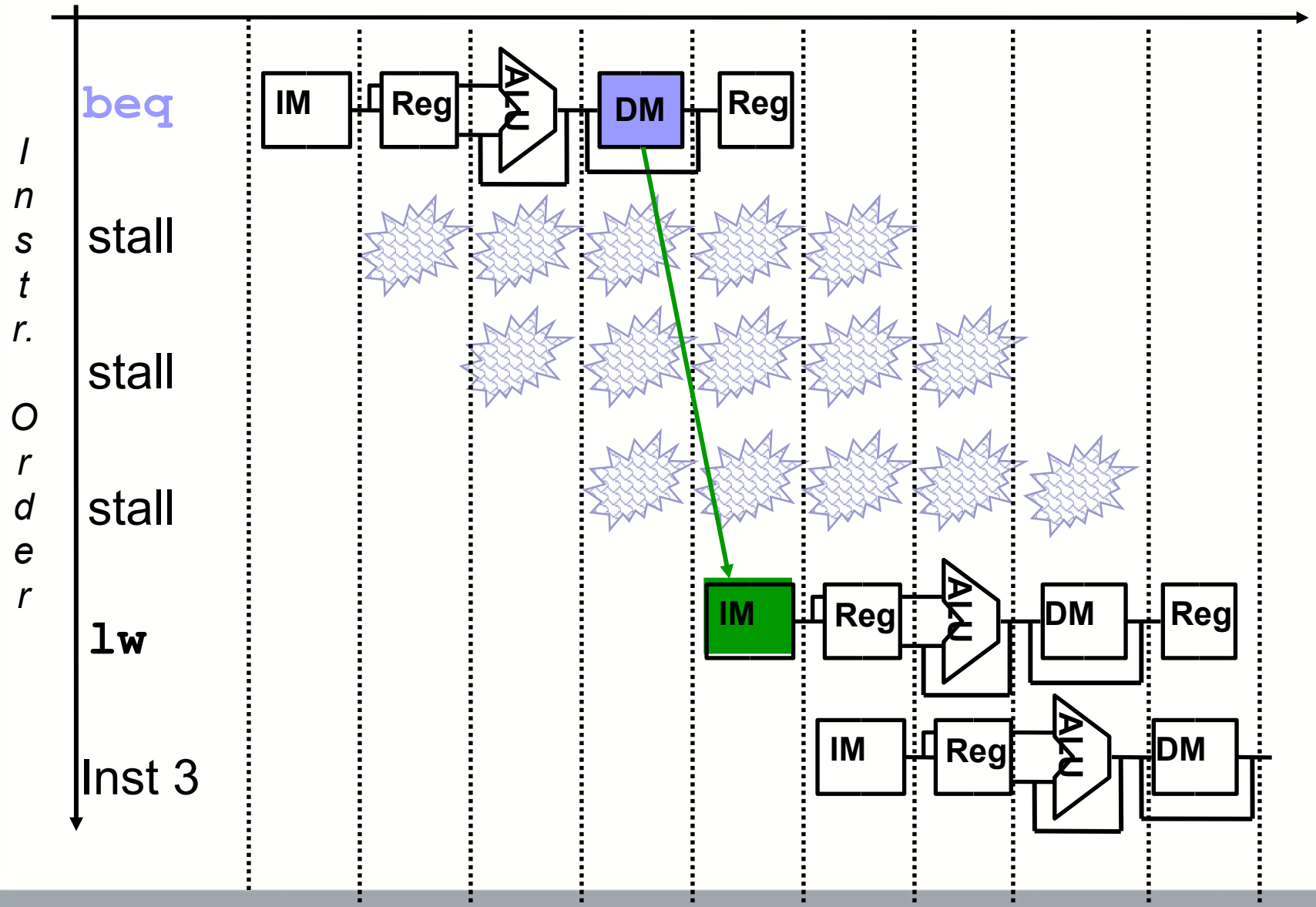
Solutions

- Possible approaches
  - Stall
  - Move decision point as early in the pipeline as possible, thereby reducing the number of stall cycles
  - Reoder instruction
  - Predict and hope for the best !

- Control hazards occur less frequently than data hazards, but there is *nothing* as effective against control hazards as forwarding is for data hazards

Stall

Fix branch hazard by waiting – stall – but affects CPI



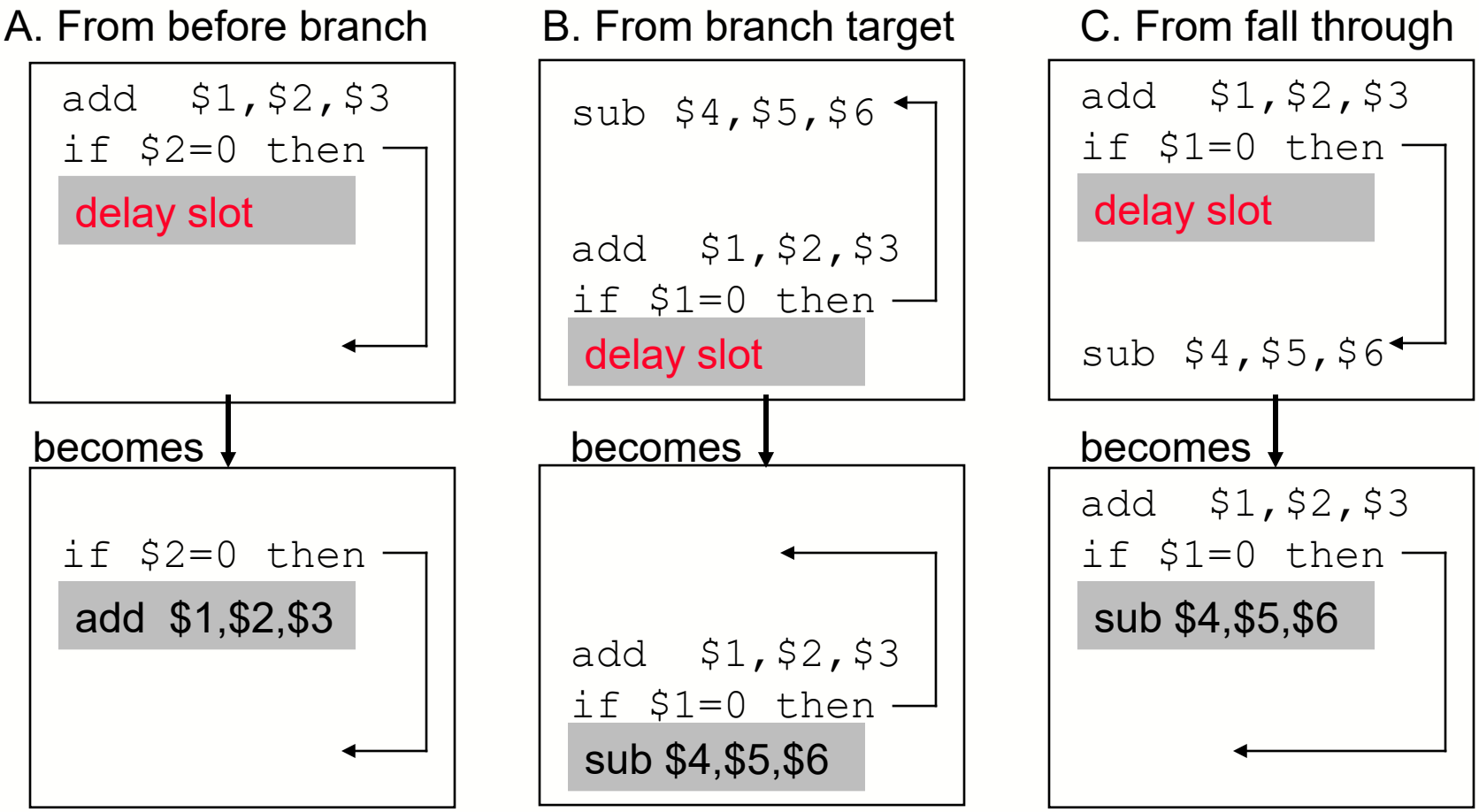
Reducing the delay of Branches

- Move the branch decision hardware back to the EX stage
  - Reduces the number of stall (flush) cycles to two
  - Adds an and gate and a 2x1 mux to the EX timing path
- Add hardware to compute the branch target address and evaluate the branch decision to the ID stage
  - Reduces the number of stall (flush) cycles to one
    - But now need to add forwarding hardware in ID stage
- For deeper pipelines, branch decision points can be even *later* in the pipeline, incurring more stalls

Delayed Decision

- If the branch hardware has been moved to the ID stage, then we can eliminate all branch stalls with delayed branches which are defined as always executing the next sequential instruction after the branch instruction – the branch takes effect *after* that next instruction
  - MIPS compiler moves an instruction to immediately after the branch that is not affected by the branch (a safe instruction) thereby hiding the branch delay
- With deeper pipelines, the branch delay grows requiring more than one delay slot
  - Delayed branches have lost popularity compared to more expensive but more flexible (dynamic) hardware branch prediction
  - Growth in available transistors has made hardware branch prediction relatively cheaper

Scheduling Branch Delay Slots



- A is the best choice, fills delay slot and reduces IC
- In B and C, the `sub` instruction may need to be copied, increasing IC
- In B and C, must be okay to execute `sub` when branch fails

Static Branch Prediction

- Resolve branch hazards by assuming a given outcome and proceeding without waiting to see the actual branch outcome

1. Predict not taken – always predict branches will not be taken, continue to fetch from the sequential instruction stream, only when branch *is* taken does the pipeline stall
   - If taken, flush instructions after the branch (earlier in the pipeline)
     - in IF, ID, and EX stages if branch logic in MEM – three stalls
     - In IF and ID stages if branch logic in EX – two stalls
     - in IF stage if branch logic in ID – one stall
   - ensure that those flushed instructions haven't changed the machine state – automatic in the MIPS pipeline since machine state changing operations are at the tail end of the pipeline (MemWrite (in MEM) or RegWrite (in WB))
   - restart the pipeline at the branch destination

Prediction

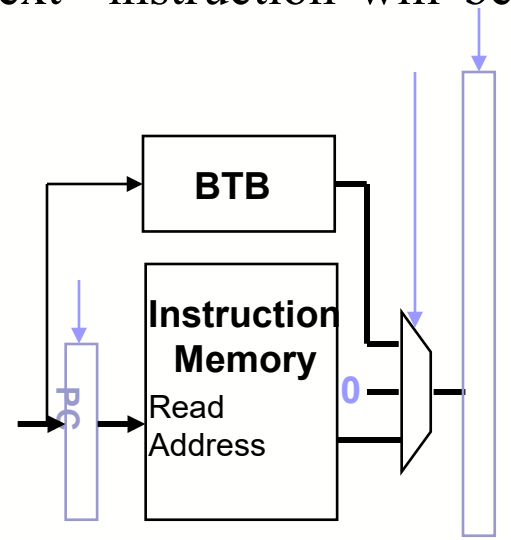
- One simple approach is to always predict that branches will fail
  - If it's right, the pipeline will proceed at full speed
  - If it's wrong, next 3 instructions will be discarded
- A more sophisticated approach is keeping a history for each branch as taken or untaken and then using the past to predict the future
  - One implementation of this approach is a ***branch prediction buffer*** or ***branch history table***

Branch Prediction Buffer

- is addressed by the lower bits of the PC
- contains a bit passed to the ID stage through the IF/ID pipeline register that tells whether the branch was taken the last time it was execute
  - Prediction bit may predict incorrectly (may be a wrong prediction for this branch this iteration or may be from a different branch with the same low order PC bits) but the doesn't affect correctness, just performance
    - Branch decision occurs in the ID stage after determining that the fetched instruction is a branch and checking the prediction bit
  - If the prediction is wrong, flush the incorrect instruction(s) in pipeline, restart the pipeline with the right instruction, and invert the prediction bit

Branch Target Buffer

- The BHT predicts *when* a branch is taken, but does not tell *where* its taken to!
  - A branch target buffer (BTB) in the IF stage can cache the branch target address, but we also need to fetch the next sequential instruction. The prediction bit in IF/ID selects which "next" instruction will be loaded into IF/ID at the next clock edge
    - Would need a two read port instruction memory
  - Or the BTB can cache the branch taken instruction while the instruction memory is fetching the next sequential instruction



- If the prediction is correct, stalls can be avoided no matter which direction they go

1-bit Prediction Accuracy

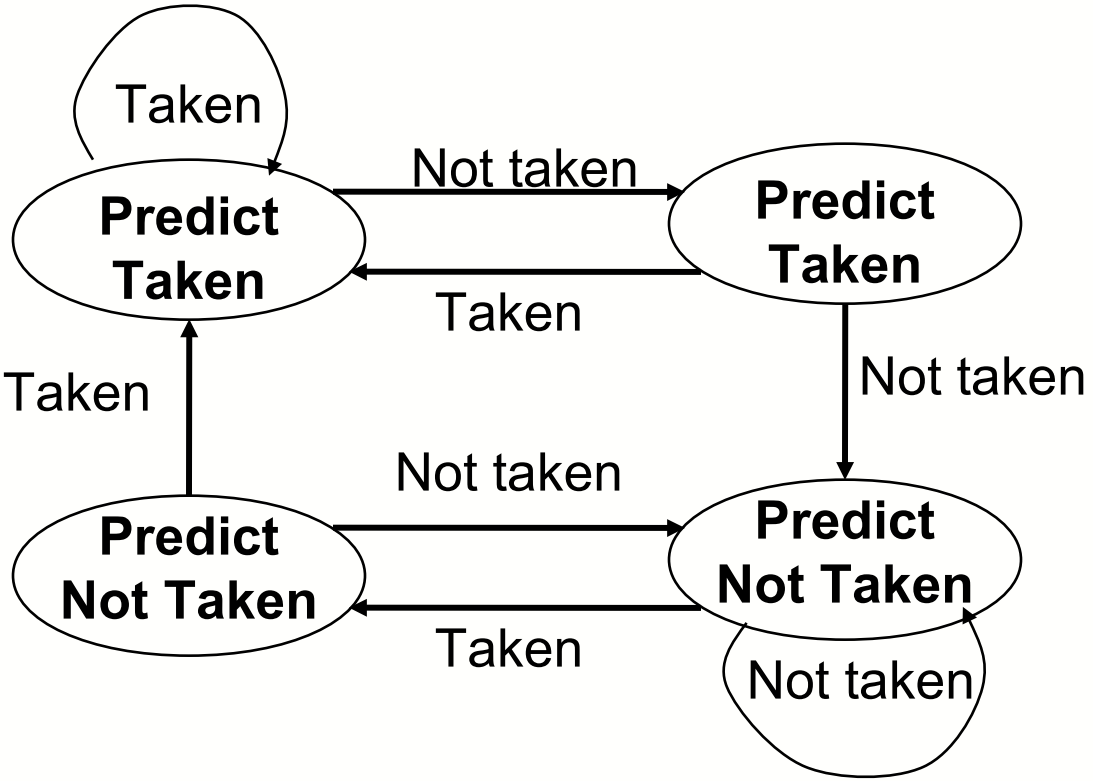
- A 1-bit predictor will be incorrect twice when not taken
  - Assume predict_bit = 0 to start (indicating branch not taken) and loop control is at the bottom of the loop code
  1. First time through the loop, the predictor mispredicts the branch since the branch is taken back to the top of the loop; invert prediction bit (predict_bit = 1)
  2. As long as branch is taken (looping), prediction is correct
  3. Exiting the loop, the predictor again mispredicts the branch since this time the branch is not taken falling out of the loop; invert prediction bit (predict_bit = 0)

```
Loop: 1st loop instr
      2nd loop instr
            .
            .
            .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

- For 10 times through the loop we have a 80% prediction accuracy for a branch that is taken 90% of the time

2-Bit Predictors

- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed




```
Loop: 1st loop instr
      2nd loop instr
            .
            .
            .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```