Branch Prediction

- Static Branch Prediction
- Dynamic Branch Prediction
- Note: Assignment 1: due Feb 19th.

Static Branch Prediction

- Compiler determines whether branch is likely to be taken or likely to be not taken.
  - How?

When is a branch likely to be taken?

When is a branch likely to be NOT taken?

```
int gtz=0;
int i = 0;

while (i < 100) {
  x = a[i];
  if (x == 0)
    continue;
  gtz++;
}
```

Static Branch Prediction

- Compiler determines whether branch is likely to be taken or likely to be not taken.
- Decision is based on analysis or profile information
  - 90% of backward-going branches are taken
  - 50% of forward-going branches are not taken
  - BTFN: “backwards taken, forwards not-taken”
  - Used in ARC 600 and ARM 11
- Decision is encoded in the branch instructions themselves
  - Uses 1 bit: 0 => not likely to branch, 1=> likely to branch
- Prediction may be wrong!
  - Must kill instructions in the pipeline when a bad decision is made
  - Speculatively issued instructions must not change processor state

Dynamic Branch Prediction

- Monitor branch behavior and learn
  - Key assumption: past behavior indicative of future behavior
- Predict the present (current branch) using learned history
- Identify individual branches by their PC or dynamic branch history
- Predict:
  - Outcome: taken or not taken
  - Target: address of instruction to branch to
- Check actual outcome and update the history
- Squash incorrectly fetched instructions

Simplest dynamic predictor: 1-bit Prediction

- 1 bit indicating Taken (1) or Not Taken (0)

- Branch prediction buffers:
  - Match branch PC during IF or ID stages

```
…
0x135c4: add r1,r2,r3
0x135c8: bne r1,r0,n
…
```

Branch PC	Outcome
0x135c8	0
0x147e0	1
…	…

- Incurs at least 2 mis-predictions per loop

```
while (i < 100) {
  x = a[i];
  if (x == 0)
    continue;
  gtz++;
}
```

Problem: "unstable" behavior

2-bit (Bimodal) Branch Prediction

- Idea: add hysteresis
  - Prevent spurious events from affecting the most likely branch outcome

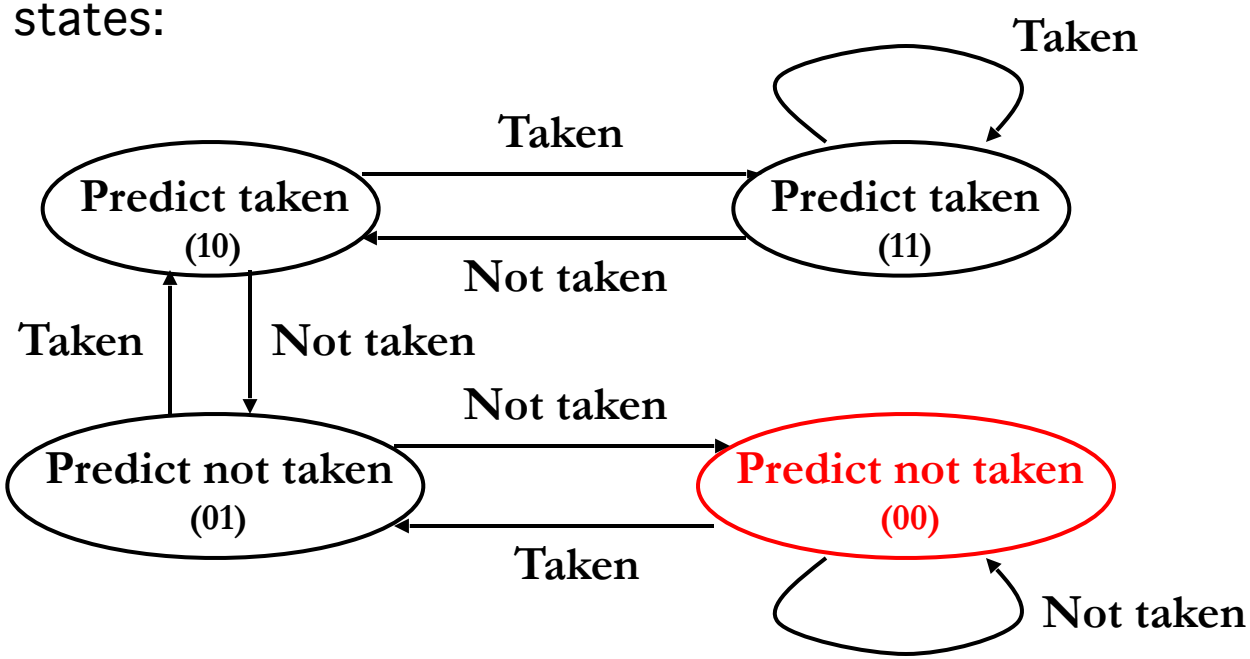
- 2-bit saturating counter:
  - 00: do not take
  - 01: do not take
  - 10: take
  - 11: take

Branch PC	Outcome
0x135c8	10
0x147e0	11
…	…

```
while (i < 100) {
  x = a[i];
  if (x == 0)
    continue;
  gtz++;
}
```

2-bit (Bimodal) Branch Prediction

- Predictor states:

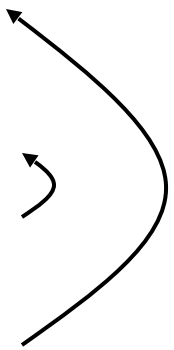


- Learns biased branches
- N-bit predictor:
  - Increment on Taken outcome and decrement on Not Taken outcome
  - If counter>$(2^n-1)/2$ then take, otherwise do not take
  - Takes longer to learn, but sticks longer to the prediction

Example of 2-bit (Bimodal) Branch Prediction

- Nested loop:

```
Loop1: …
       …
Loop2: …
       bne r1,r0,loop2
       …
       bne r2,r0,loop1
```



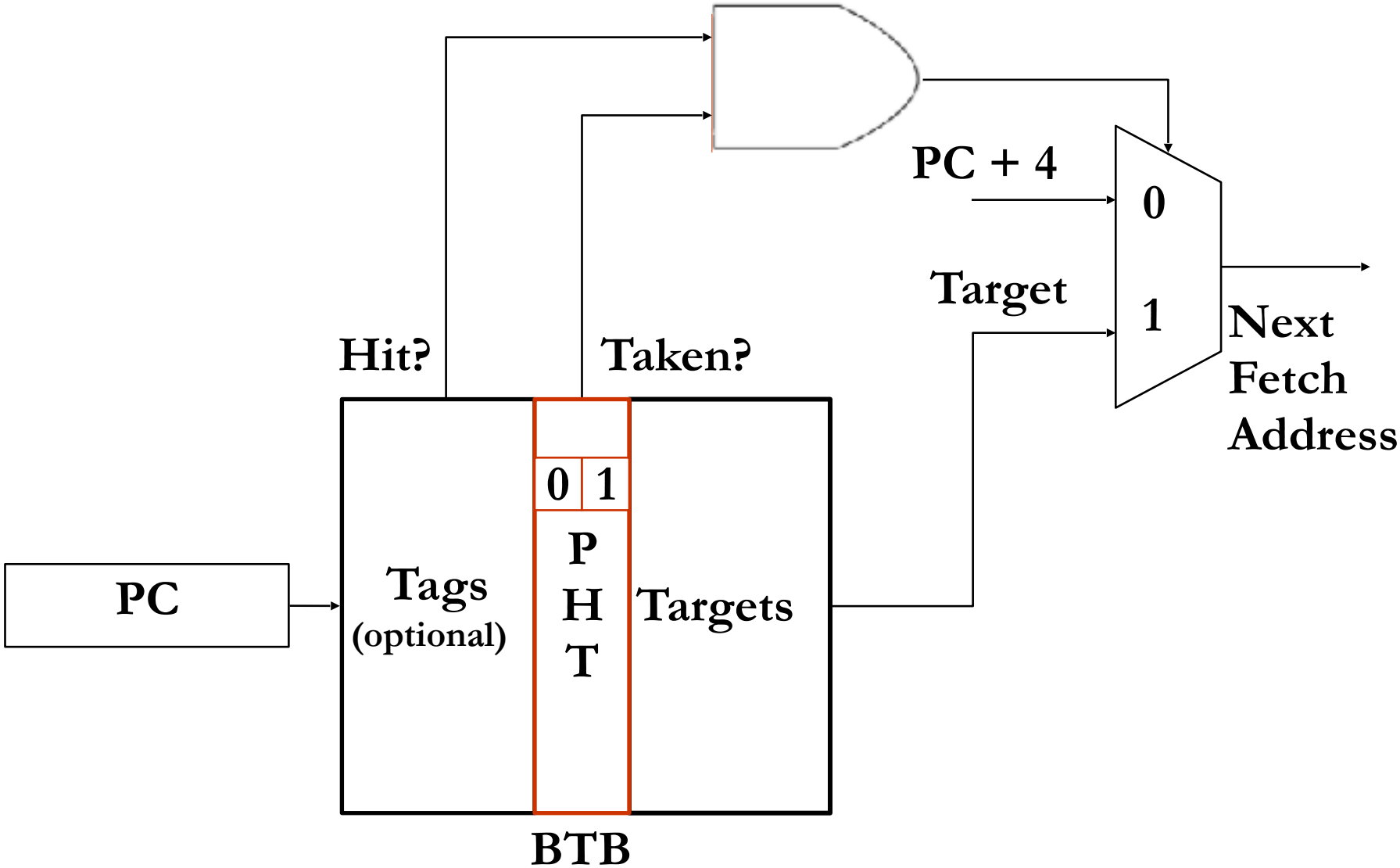
- 1st outer loop execution:
  - 00 → predict not taken; actually taken → update to 01 (misprediction)
  - 01 → predict not taken; actually taken → update to 10 (misprediction)
  - 10 → predict taken; actually taken → update to 11
  - 11 → predict taken; actually taken
  - …
  - 11 → predict taken; actually not taken → update to 10 (misprediction)

Example Continued

- 2nd outer loop execution onwards:
  - 10 → predict taken; actually taken → update to 11
  - 11 → predict taken; actually taken
  - ...
  - 11 → predict taken; actually not taken → update to 10 (misprediction)
- In practice misprediction rates for 2-bit predictors with 4096 entries in the buffer range from 1% to 18% (higher for integer applications than for fp applications)

- Bottom-line: 2-bit branch predictors work very well for loop-intensive applications
  - n-bit predictors (n>2) are not much better
  - Larger buffer sizes do not perform much better

Bimodal (2-bit) predictor logic



Correlating Predictors

- 1- and 2-bit predictors exploit most recent history of the current branch
- Realization: branches are correlated!
  - Local: A branch outcome maybe correlated with past outcomes (multiple outcomes or history, not just the most recent) of the same branch
  - Global: A branch outcome maybe correlated with past outcomes of other branches

Correlating Predictors (Local)

- 1- and 2-bit predictors exploit most recent history of the current branch
- Realization: outcomes of same branch correlated!
-

```
while (i < 4) {
  x = a[i];
  if (x == 0)
    continue;
  gtz++;
}
```

- Branch outcomes: 1,1,1,0, 1,1,1,0 1,1,1,0....

Idea: exploit recent history of <u>same</u> branch in prediction

Correlating Predictors (Global)

- 1- and 2-bit predictors exploit most recent history of the current branch
- Realization: Different branches maybe correlated!

```
if (a == 2)
    a = 0;
if (b == 2)
    b = 0;
if (a != b) {
    …}
```

If both branches are taken,
the last branch definitely <u>not taken</u>

```
char s1 = "Bob"
...
if (s1 != NULL)
  reverse_str(s1);
  ____________
reverse_str(char *s) {
  if (s1 == NULL)
    return;
 ...
```

s1 definitely <u>not</u> Null
<u>in this calling context</u>

Correlating Predictors (Global)

- 1- and 2-bit predictors exploit most recent history of the current branch
- Realization: Different branches maybe correlated!

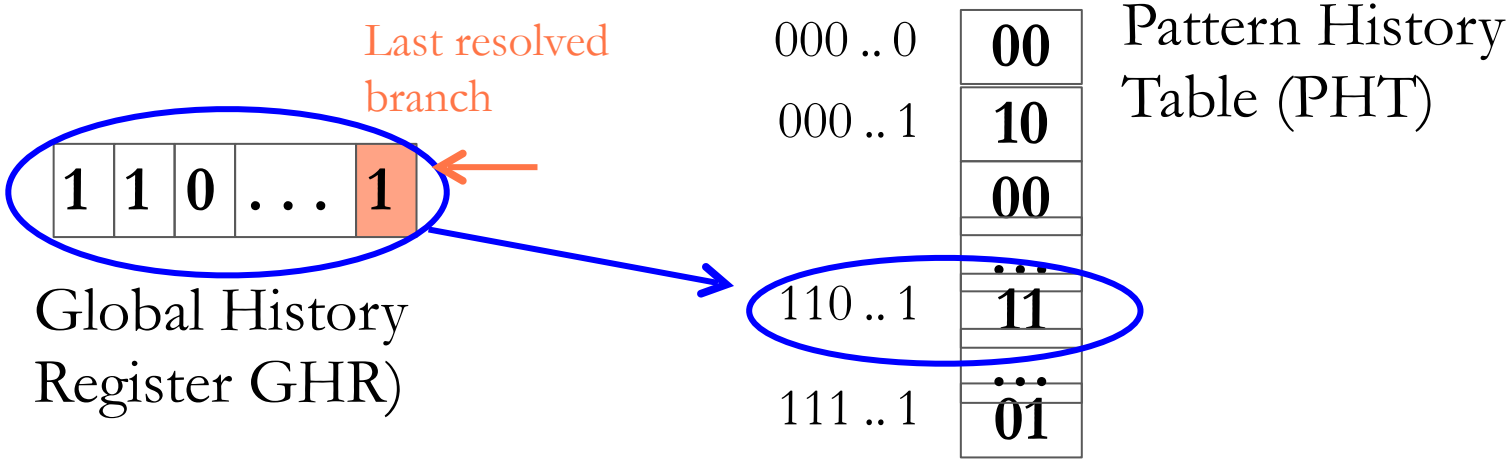
```
if (a == 2)
    a = 0;
if (b == 2)
    b = 0;
if (a != b) {
    …}
```

```
char s1 = "Bob"
...
if (s1 != NULL)
  reverse_str(s1);
  ___________
reverse_str(char *s) {
  if (s1 == NULL)
    return;
 ...
```

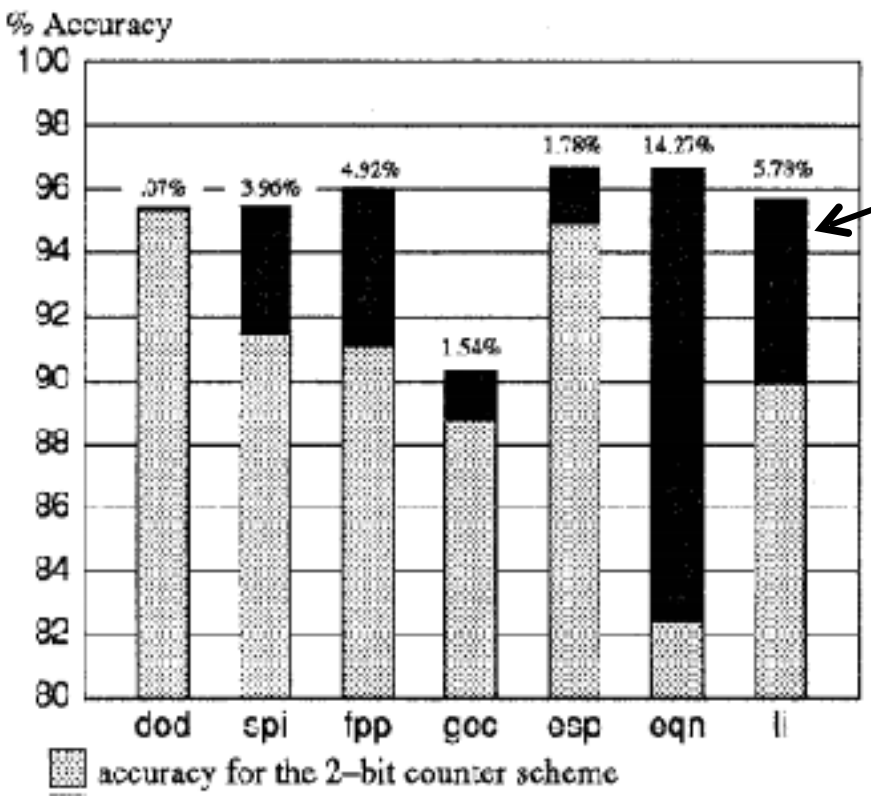
Idea: exploit recent history of <u>other</u> branches in prediction

Global Two-Level (or Correlating) Predictor

- Prediction depends on the <u>context</u> of the branch
- Context: history (T/NT) of the last N branches
  - First level of the predictor
  - Implemented as a shift register
- Prediction: 2-bit saturating counters
  - Indexed with the "global" history
  - Second level of the predictor



Dynamic Predictor Performance Comparison



2-level correlating predictor with a 15-bit global history

2-level predictor improves accuracy by >4%

Does 4% accuracy improvement matter?

- Assume branches resolve in stage 10
  - Reasonable for a modern high-frequency processor
- 20% of instructions are branches
- Correctly-predicted branches have a 0-cycle penalty (CPI=1)
- 2-bit predictor: 92% accuracy
- 2-level predictor: 96% accuracy

<u>2-bit predictor:</u>

$$CPI = 0.8 + 0.2 * (10*0.08 + 1*0.92) = 1.114$$

<u>2-level predictor</u>

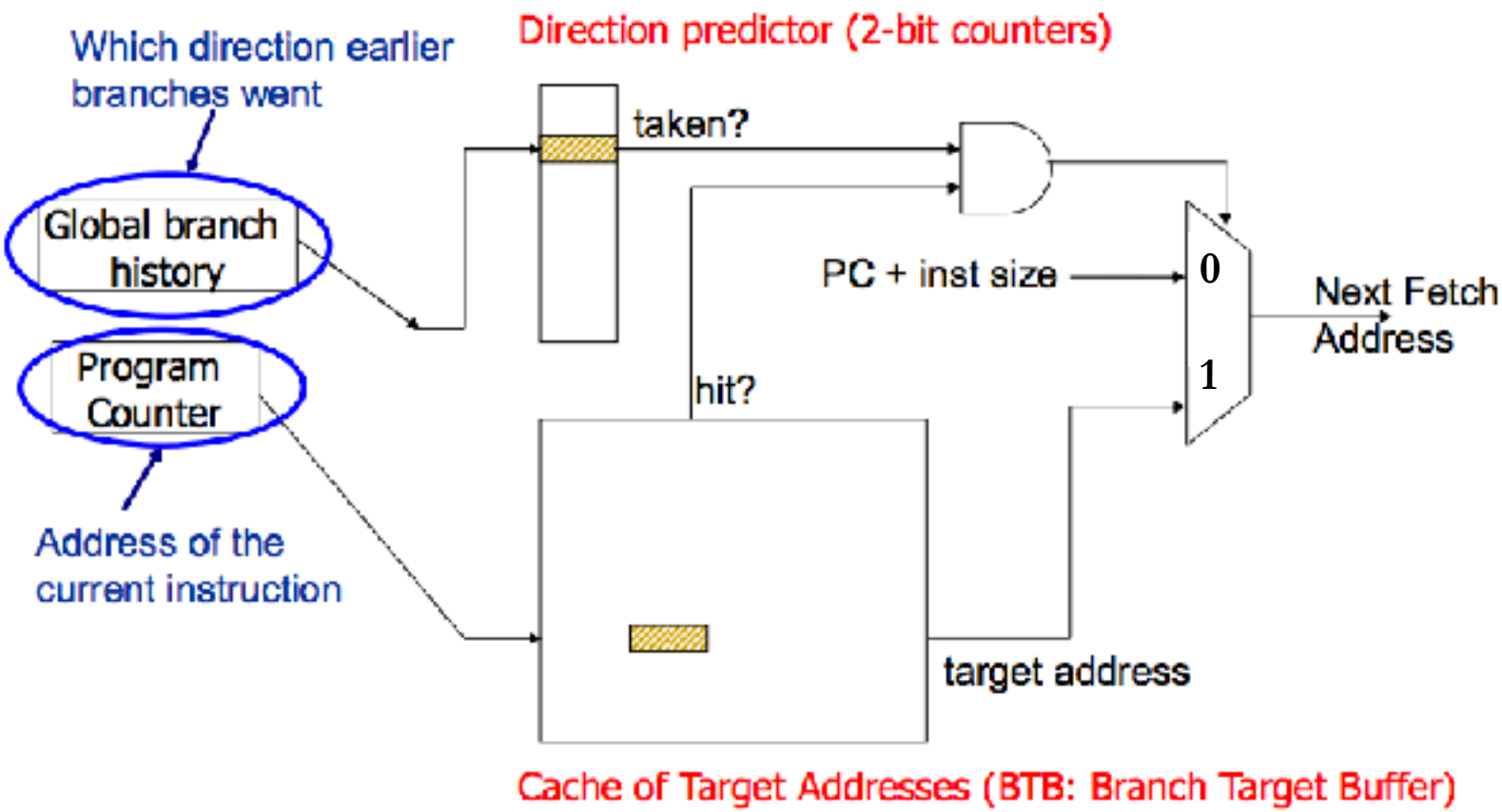
$$CPI = 0.8 + 0.2 * (10*0.04 + 1*0.96) = 1.072$$

Speedup(2-level over 2-bit): 4%

Branch Target Buffers (BTB)

- Branch predictors tell whether the branch will be taken or not, but they say nothing about the target of the branch
- To resolve a branch early we need to know both the outcome and the target
- Solution: store the likely target of the branch in a table (cache) indexed by the branch PC → BTB
- Usually BTB is accessed in the IF stage and the branch predictor is accessed later in the ID stage

Branch Prediction Logic of global 2-level predictor



Source: Onur Mutlu, CMU

Bimodal and Global 2-level

- Bimodal (2-bit) Branch Predictor

  + Good for biased branches

  + No interference

  - Cannot discern patterns

- Global 2-level Branch Predictor

  + Leverages correlated branches

  + Identifies patterns

  - Cannot always take advantage of biased branches

  - Interference

Interference in Global 2-level Predictors

Pattern History Table

Global History Register

1	0	1

1	1	0
0	1	1
1	1	2
0	1	3
1	1	4
1	0	5
1	1	6
0	1	7

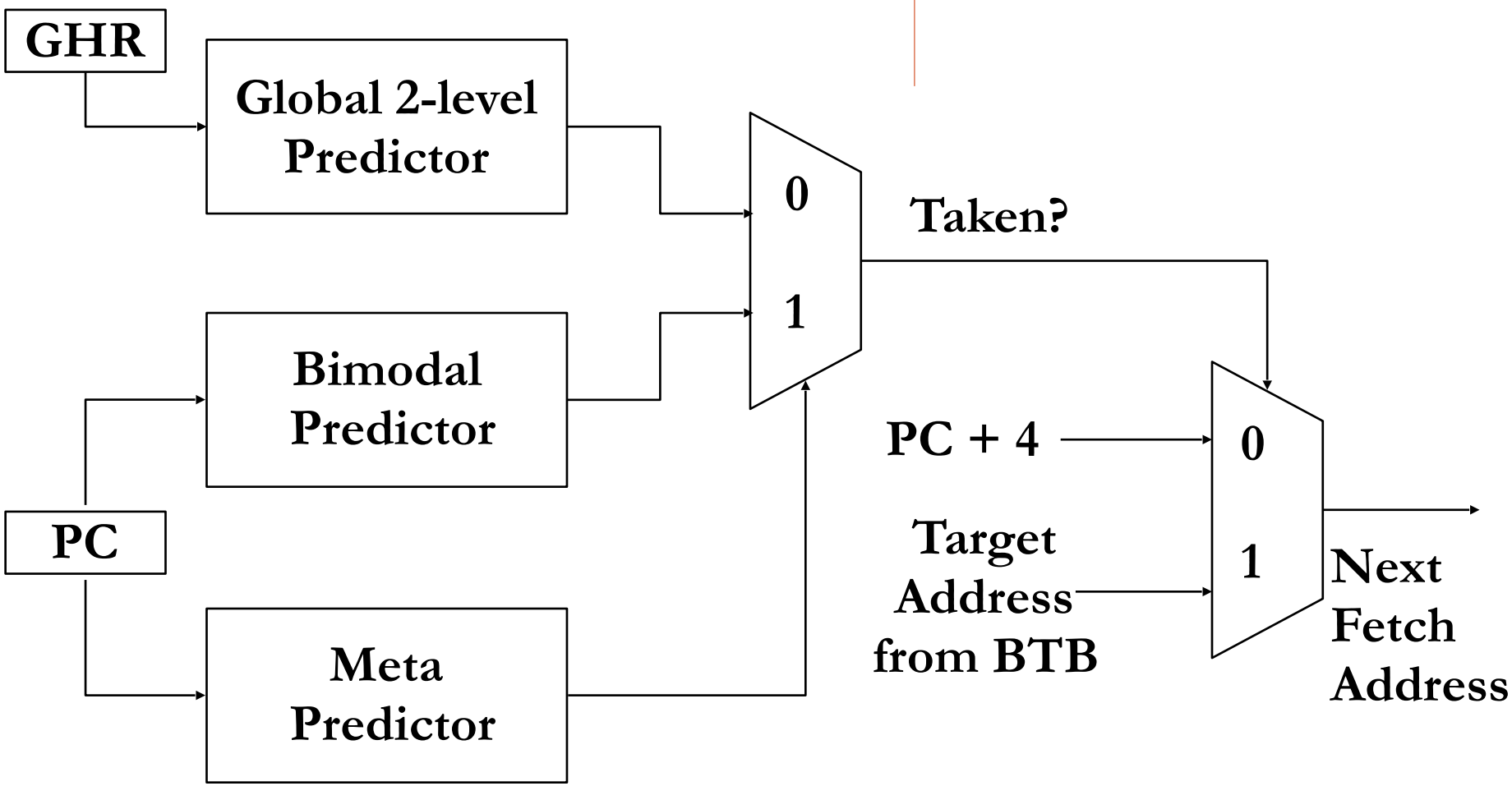
- **Branch A is always Not Taken when GHR is 101**
- **Branch B is a loop with a million iterations**
- **Branch A and Branch B can interfere in entry 5 of the PHT**

Biased branches pollute the PHT!!!!

Tournament Predictor - the best of multiple predictors

- Most branches are biased (e.g. 99% Taken)
- Filter the biased branches with a simple predictor (e.g. Bimodal)
- Predict the hard branches with the Global 2-level predictor
- Use a meta-predictor to chose a different predictor
- The meta-predictor is a PHT of 2-bit saturating counters

Tournament Predictor



References

- Different solutions to the problem of interference
  - Gshare – Use a hash function to index to the PHT (CAR assignment uses this as the “global” predictor)
- What is the state of the art ?
  - TAGE (Use multiple tagged PHTs for multiple history lengths)
  - Perceptron (Learn the correlations in the global history)
- Branch Prediction Championship
  - https://www.jilp.org/cbp2016/

Assignment 1

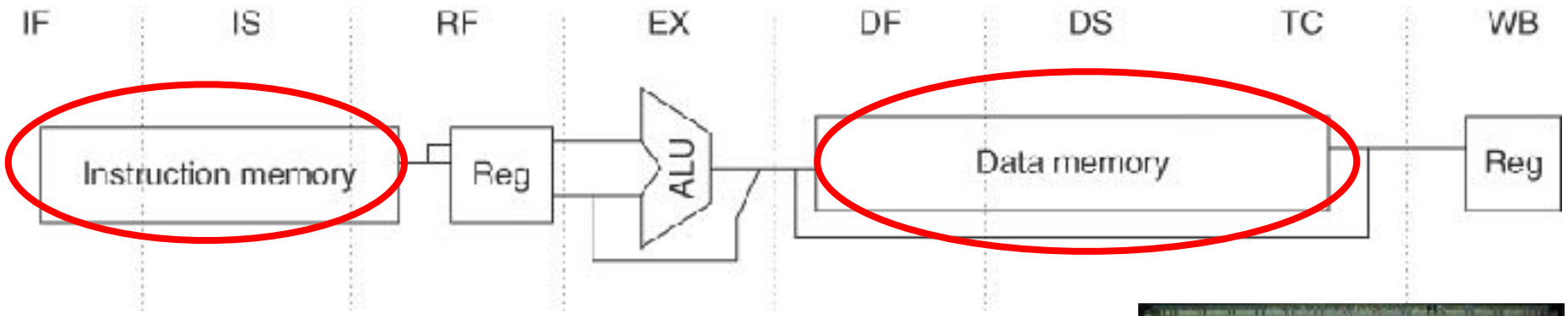
- Implement with Pin in C++
  - 2-level Local Brach predictor
  - 2-level Global Branch Predictor (gshare)
  - Tournament Branch Predictor
- BTB not required
- Correctness testing is your responsibility
  - Come up with simple micro-benchmarks with branch outcomes that you can reason about
  - Run them through your predictors and verify outcomes

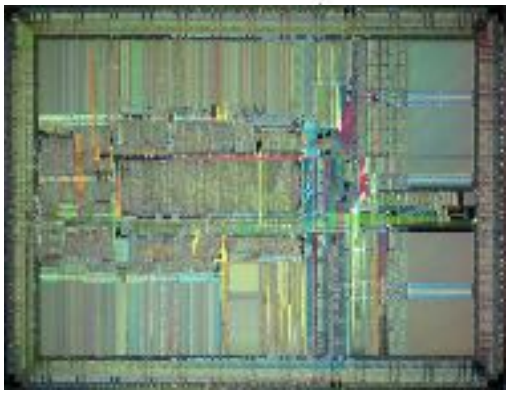
Due: Monday, Feb 19, 4pm

Handling Hazards

- Structural hazards
  - Stalling: pipeline interlock
  - Code scheduling
- Data hazards
  - Stalling: pipeline interlock
  - Forwarding
  - Load delay
    - Stalling: pipeline interlock
    - Code scheduling: fill the load delay slot
- Control Hazards
  - Early branch resolution
  - Stalling: flushing the pipeline
  - Delayed branch
  - Predict non taken (or taken)
  - Static branch prediction
  - Dynamic branch prediction

Hazards caused by multi-cycle operations

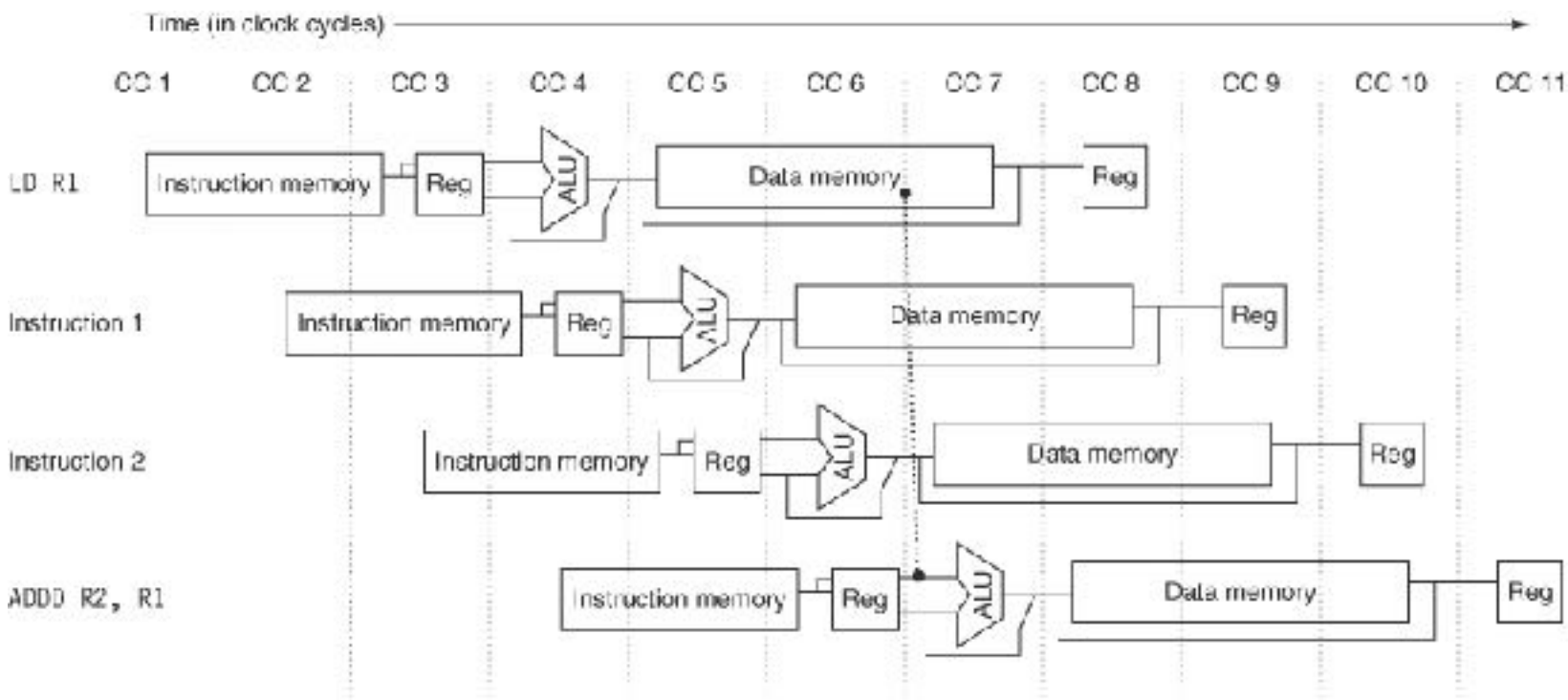




Example of this in the MIPS R4000

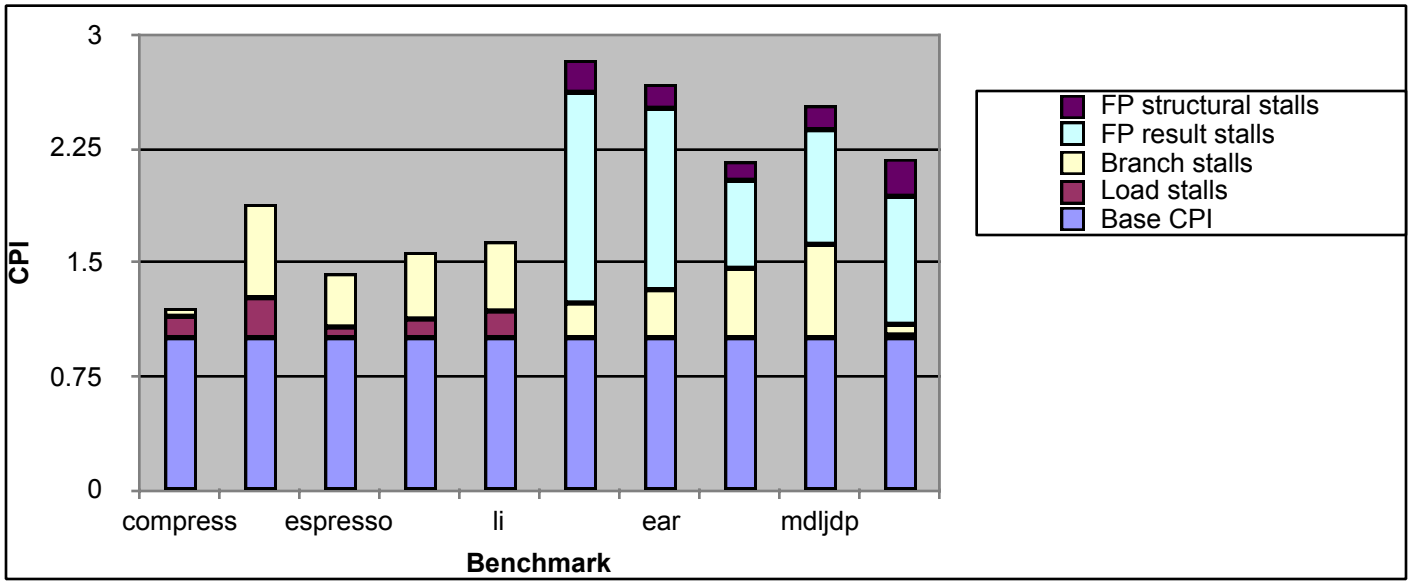
- Notable feature:
  pipelined memory accesses

Load-to-use latency in the MIPS R4000



2-cycle load delay slot

Impact of Empty Load-delay Slots on CPI

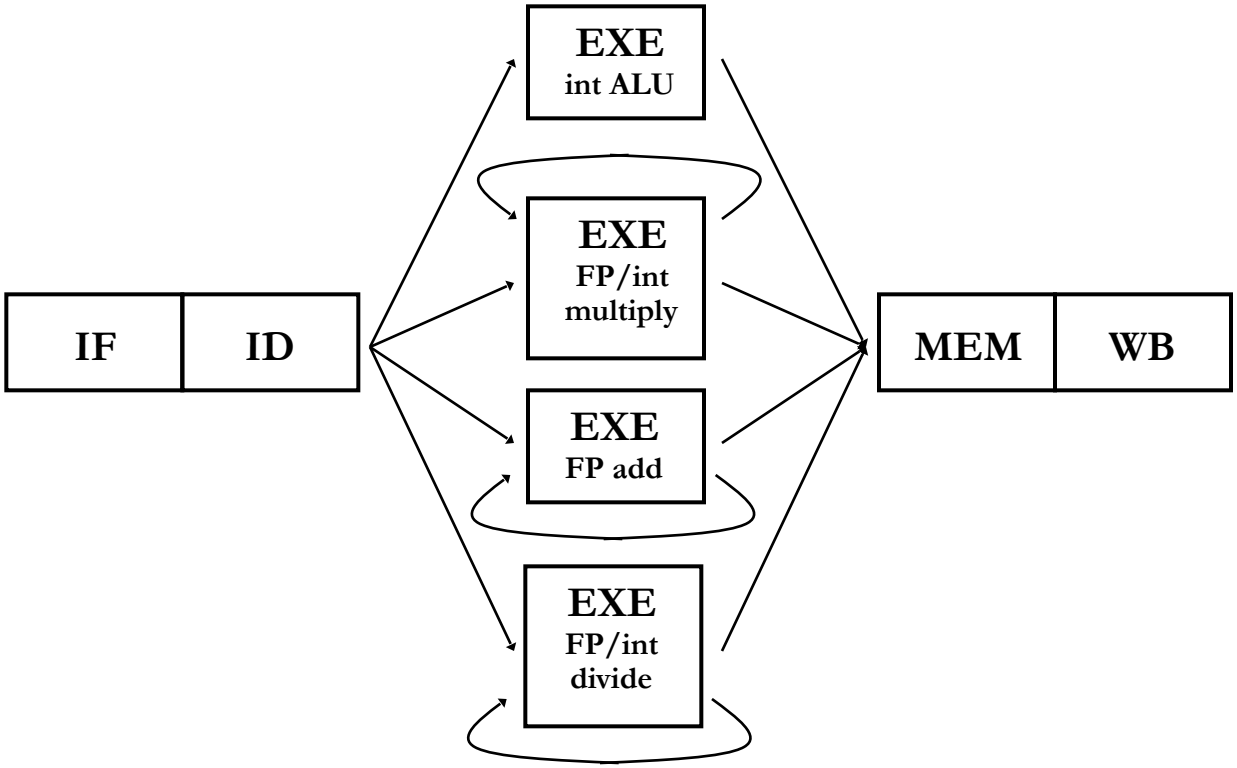


H&P 5/e
Fig. C.52

Bottom-line: CPI increase of 0.01 – 0.27 cycles

Multicycle Floating Point Operations

- Floating point operations take multiple cycles in EXE
- Example system: 1 int ALU, 1 FP/int multiplier, 1 FP adder, 1 FP/int divider



Generalizing Multicycle Operations

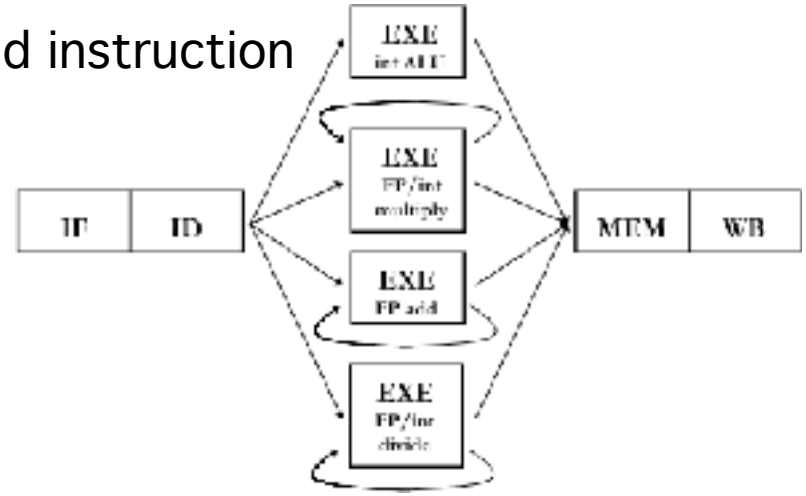
- Instruction latency: cycles to wait for the result of an instruction
  - Usually the number of cycles for the execution pipeline minus 1
  - e.g. 0 cycles for integer ALU since no wait is necessary
- Instruction initiation interval: time to wait to issue another instruction of the same type
  - Not equal to number of cycles, if multicycle operation is pipelined or partially pipelined
- Examples:
  - Integer ALU:
    1 EXE cycle → latency = 0; initiation interval = 1
  - FP add, fully pipelined:
    4 EXE cycles → latency = 3; initiation interval = 1
  - FP divide, not pipelined:
    25 EXE cycles → latency = 24; initiation interval = 25

Multicycle Functional Units: MIPS R4000

- ALU: 64-bit, fully pipelined
- Barrel shifter: 32-bit, 1-cycle pipeline stall on 64-bit shifts
  - This design was adopted to save chip area
- Integer Multiplier: not pipelined,
  10-cycle (32-bit) or 20-cycle (64-bit) latency
- Integer Divider: not pipelined,
  69-cycle (32-bit) or 133-cycle (64-bit) latency
- FP adder/multiplier: fully pipelined
- FP divider: 23- (sp) to 36-cycle (dp) latency

Multicyle Operations: Handling Hazards

- Structural hazards can occur when functional unit not fully pipelined (initiation interval > 1) → need to add interlocking
  - Stalls for hazards become longer and more frequent
- Possibly more than one register write per cycle → either add ports to register file or treat conflict as a hazard and stall
- Possible hazards between integer and FP instructions → use separate register files
- WAW hazards are possible → stall second instruction or prevent first instruction from writing



Loop Example

for (i=1000; i>0; i--)
x[i] = x[i] + s

- Straightforward code and schedule:
  - Assume **F2** contains the value of s
  - Load latency equals 1
  - FP ALU latency 3 to another FP ALU and 2 to a store

```
loop: L.D     F0,0(R1)     ;F0=array element
      stall                ;add depends on ld
      ADD.D   F4,F0,F2     ;main computation
      stall                ;st depends on add
      stall
      S.D     F4,0(R1)     ;store result
      ADDUI   R1,R1,-8     ;decrement index
      stall                ;bne depends on add
      BNE     R1,R2,loop   ;next iteration
      stall                ;branch delay slot
```

Cycle
1
2
3
4
5
6
7
8
9
10

Iteration Scheduling

- Execution time of straightforward code: 10 cycles/element
- Smart compiler (or human ☺) schedule:

```
loop: L.D     F0,0(R1)     ;F0=array element
      DADDUI  R1,R1,-8     ;decrement index
      ADD.D   F4,F0,F2     ;main computation
      stall                ;st depends on add
      BNE     R1,R2,loop   ;next iteration
      S.D     F4,8(R1)     ;store result
```

Cycle
1
2
3
4
5
6

- Immediate offset of store was changed after reordering
- Execution time of scheduled code:

  6 cycles/element → Speedup=1.7

- Of the 6 cycles, 3 are for actual computation (l.d, add.d, s.d), 2 are loop overhead (addi, bne), and 1 is a stall