Resolving Data Hazards by Stalling

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>sub</td>
<td>sub</td>
<td>sub</td>
<td>xori</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>addi</td>
<td>xor</td>
<td>xor</td>
<td>xor</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>addi</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>xori</td>
<td>sub</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>addi</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>addi</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stalls increase CPI!

Resolving Data Hazards by Bypassing

- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated.

- addi writes to x11 at the end of cycle 5... but the result is produced during cycle 3, at the EXE stage!

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>addi</td>
<td>xor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>addi</td>
<td>addi</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

addi result computed -> x11 updated
Load-To-Use Stalls

- Bypassing cannot eliminate load delays because their data is not available until the WB stage
  
  - Bypassing from WB still saves a cycle:
  
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>lw</td>
<td>xor</td>
<td>sub</td>
<td>sub</td>
<td>sub</td>
<td>xori</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>lw</td>
<td>xor</td>
<td>xor</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>xori</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>xori</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- lw data available
- x11 updated

![Pipeline Diagram]

6.004 Worksheet - 2 of 11 - L19 – Data Hazards
Problem 1. ★

The program shown on the right is executed on a 5-stage pipelined RISC-V processor with full bypassing.

The program has been running for a while and execution is halted at the end of cycle 105.

The pipeline diagram shown below shows the history of execution at the time the program was halted.

(A) Please indicate on which cycle(s), 100 through 105, each of the following actions occurred. If the action did not occur in any cycle, write “NONE”.

<table>
<thead>
<tr>
<th>cycle</th>
<th>100</th>
<th>101</th>
<th>102</th>
<th>103</th>
<th>104</th>
<th>105</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>slli</td>
<td>lw</td>
<td>add</td>
<td>bnez</td>
<td>bnez</td>
<td>bnez</td>
</tr>
<tr>
<td>DEC</td>
<td>add</td>
<td>slli</td>
<td>lw</td>
<td>add</td>
<td>add</td>
<td>add</td>
</tr>
<tr>
<td>EXE</td>
<td>NOP</td>
<td>addi</td>
<td>slli</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>MEM</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td>slli</td>
<td>lw</td>
<td>NOP</td>
</tr>
<tr>
<td>WB</td>
<td>bnez</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td>slli</td>
<td>lw</td>
</tr>
</tbody>
</table>

Register value used from Register File: ______ 100, 105____

Register value bypassed from EXE stage to DEC stage: _____ 101, 102____

Register value bypassed from MEM stage to DEC stage: ______NONE____

Register value bypassed from WB stage to DEC stage: _______ 105_____

In cycle 101, decoding the slli instruction requires the value of x11. However, x11 was modified by the addi instruction immediately before it. Therefore, we cannot use the value of x11 that is currently in the register file since that will not be updated until the end of cycle 103 when the addi instruction goes through the write-back stage. However, we can get the value that will ultimately be stored in x11 by looking at the result of executing the addi instruction (i.e. bypassing the result of the EXE stage to the DEC stage, EXE → DEC bypass).

Similarly, decoding the lw instruction in cycle 102 requires the updated value in x13, which is modified by the slli instruction. Once again, we can bypass from the EXE stage, where the addi instruction has now passed through the ALU (EXE → DEC bypass).
In cycle 103, we attempt to decode the add instruction in the DEC stage, but this requires knowing x12 and x14. We can get x12 from the register file, but x14 was updated by a lw operation. The lw operation cannot actually get the data from memory until the start of the WB stage, and it doesn’t update the x14 register until the end of the WB stage. At cycle 103, the lw instruction has not reached the WB stage yet. We must insert NOPs into the pipeline until the lw hits the WB stage (cycle 105), at which point we can bypass the value retrieved from memory back into the DEC stage (WB \rightarrow DEC bypass). The value of x12 can just be retrieved from the register file.

(B) Why is the NOP instruction inserted in cycle 104?

The add must wait until the lw is in the WB stage in order to be able to get the results of the lw via a bypass path. See previous part for a more detailed explanation.
Problem 2. ★

The following program fragments are being executed on the 5-stage pipelined RISC-V processor with full bypassing. For each fragment, the pipeline diagram shows the state of the pipeline for cycle 1000 of execution. Please fill in the diagram for cycle 1001; use “?” if you cannot tell what opcode to write into a stage. Then for both cycles use arrows to indicate any bypassing from the EXE/MEM/WB stages back to the DEC stage.

(A)

```
... sw x1, 0(x0)
lw x17, 0xC(x1)
addi x2, x2, -4
slli x11, x17, 2
sw x11, 0(x2)
jal ra, fact
...
```

To figure out if we need to add a NOP, we look at the instruction we are decoding: slli x11, x17, 2. Not all of these values are available to us by cycle 1000. Specifically, x17 was modified by the lw instruction. Notice from the pipeline diagram that lw is still in the MEM stage at cycle 1000, meaning that it has just sent a request to the data memory. However, the data memory does not return the data until one clock cycle later. Therefore, we need to insert a NOP to wait for the lw to reach the WB stage. At the end of cycle 1001, the lw has reached the WB stage which means the data has been retrieved from memory and can be bypassed up to the DEC instruction to decode the slli instruction that has been stalled there for one cycle.

(B)

```
... xor x11, x11, x12
slli x12, x12, 3
sub x13, x12, x11
and x13, x13, x11
add x13, x12, x13
sw x13, 0x100(x0)
...
```

In cycle 1000, the and instruction is being decoded in the DEC stage. The and instruction relies on registers x11 and x13. Both of these registers were updated within 3 cycles of the and instruction, so the results of these registers have not yet been written back into the register files. The value of x13 is set by the sub instruction which is currently in the EXE stage. Similarly, the value for x11 needs to be bypassed from the WB stage since it is set by the xor instruction.

Since all required register values could be retrieved in cycle 1000, there is no need for a NOP. We process the next instruction which is a sw instruction. The add instruction is now being decoded and requires the values in x12 and x13. These were modified by the and and sub instructions, so we need to bypass from the EXE and MEM stages.
Problem 3.

For the following questions, assume that you are running code on the 4-stage pipelined processor whose stages are shown below. Also assume that:

- there is no bypassing,
- the register file (RF) is not a bypass register file,
- there are no early writebacks (meaning that every instruction must go through the last stage),
- there is a register between pipeline stages (not a FIFO buffer), and
- data memory responds immediately (in the next clock cycle) to any load request.

The problems will ask about pipeline hazards; we will say a hazard is observable if it causes the pipeline to stall.

Note: For some problems it might be helpful to draw a pipeline diagram. There are several blank pipeline diagrams on the next page that you can use as well as additional ones on the last page of the exam.

```
0x100  addi sp, sp, -4
0x104  sri a1, a0, 1
0x108  addi a4, a4, +1
0x10c  beqz a0, L0
0x110  andi a2, a1, 0x001
0x114  L0: sub a3, a0, a2
0x118  lw a5, +4(sp)
0x11c  sw a3, 0(sp)
```

(A) (3 points) Identify all the potential read-after-write (RAW) data hazards in the code above (including ones that are not observable). The number at the beginning of each line corresponds to the address of that line of code. For each hazard, write Lines a and b if a register is written in the line at address a and read in the line at address b. If there are more spaces than you need, leave the extra spaces blank.

1. Lines __0x104____ and __0x110____ sri a1 – andi a1
2. Lines __0x110____ and __0x114____ andi a2 – sub a2
3. Lines __0x114____ and __0x11c____ sub a3 – sw a3
4. Lines __0x100____ and __0x118____ addi sp – lw sp
For each of the scenarios below, whenever you are asked to list the observable hazards, specify the number to the left of the hazard you are referring to from part A. So, if the hazards labelled (1) and (3) in part A of your answer are observable, the enter 1, 3 as your observable hazards response. Enter None if there are no observable hazards.

(B) (4 points) Which of the hazards from part A are observable when the branch (address 0x10C) is taken and is correctly predicted as being taken?

List of observable hazards or None: _3 (0x114-0x11C, sub & sw)_

See pipeline diagram on next page.

(C) (4 points) Which of the hazards from part A are observable when the branch is not taken and is correctly predicted as being not taken? How many nops are inserted (i.e., for how many cycles is the pipeline stalled) in this case?

List of observable hazards or None: _2 (0x110-0x114, andi & sub), 3 (0x114-0x11C, sub & sw)_

Number of nops/stalled cycles: _3_

See pipeline diagram on next page.

(D) (4 points) Now reconsider part (C) where the branch is not taken and is correctly predicted as being not taken. However, this time assume that there is a cache miss fetching the lw instruction at address 0x118 causing a three cycle stall. Which hazards are now observable?

List of observable hazards or None: _2 (0x110-0x114, andi & sub)_

Delay of lw fetch lets sub complete by the time sw is in the DEC stage.

(E) (5 points) Once again reconsider part (C) where the branch is not taken and is correctly predicted as being not taken. However, now suppose we add bypassing from the writeback stage to the decode stage (or equivalently, we replace the register file with a bypass register file), and the instruction cache no longer misses. Now which hazards are observable? How many nops are inserted (i.e., for how many cycles is the pipeline stalled) in this case?

List of observable hazards or None: _2 (0x110-0x114, andi & sub)_

Number of nops/stalled cycles: _1_

Bypass from WB to DEC saves one cycle for sub so only 1 NOP instead of 2.
<table>
<thead>
<tr>
<th>Branch taken</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td>srli</td>
<td>addi</td>
<td>beqz</td>
<td>sub</td>
<td>lw</td>
</tr>
<tr>
<td>DEC</td>
<td>addi</td>
<td>srli</td>
<td>addi</td>
<td>beqz</td>
<td>Sub</td>
<td>lw</td>
</tr>
<tr>
<td>EXE</td>
<td>addi</td>
<td>srli</td>
<td>addi</td>
<td>beqz</td>
<td>sub</td>
<td>lw</td>
</tr>
<tr>
<td>WB</td>
<td>addi</td>
<td>srli</td>
<td>addi</td>
<td>beqz</td>
<td>sub</td>
<td>lw</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Branch not taken</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td>srli</td>
<td>addi</td>
<td>beqz</td>
<td>andi</td>
<td>sub</td>
</tr>
<tr>
<td>DEC</td>
<td>addi</td>
<td>srli</td>
<td>addi</td>
<td>beqz</td>
<td>andi</td>
<td>sub</td>
</tr>
<tr>
<td>EXE</td>
<td>addi</td>
<td>srli</td>
<td>addi</td>
<td>beqz</td>
<td>andi</td>
<td>NOP</td>
</tr>
<tr>
<td>WB</td>
<td>addi</td>
<td>srli</td>
<td>addi</td>
<td>beqz</td>
<td>andi</td>
<td>NOP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Branch not taken 2</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>lw</td>
<td>sw</td>
<td>sw</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>sub</td>
<td>lw</td>
<td>NOP</td>
<td>sw</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>NOP</td>
<td>sub</td>
<td>lw</td>
<td>NOP</td>
<td>sw</td>
<td></td>
</tr>
</tbody>
</table>
Problem 4. (from Quiz 3, Fall 2020)

Val wants to have a Thanksgiving party, but there’s a nasty virus going around. Instead, she will buy the most expensive vegan turkey, a tomato, and a celery. She is at Whole Foods. She already knows that she can buy a vegan turkey for $10 at Wal-Mart, so she will be looking for something more expensive. Val wants to figure out how much her vegan meal will cost so she writes some C code, before converting it into assembly (shown on the right). Assume the registers are initialized to the values specified in the assembly code comments.

C Code

```c
int price[6] = {7, 5, 8, 10, 15, 7};
int maximum = 10;
int celery = 3;
int tomato = 5;
for (int i = 0; i < 6; i++) {
    if (price[i] > maximum)
        maximum = price[i];
}
int total_cost = maximum + celery + tomato;
```

Assembly Code

```
// x4 = 0x24 - length of price in bytes
// x5 = 0x3 - celery
// x6 = 0x5 - tomato
// x7 = 0xA - maximum
// x1 = 0x400 - address of price[0]

start:  addi x2, x0, 0
        slli x2, x2, 2
loop:   add  x8, x2, x1
        lw x3, 0(x8)
        bge x7, x3, skip
        mv x7, x3
        ori x7, x7, 0
skip:   addi x2, x2, 4
        blt x2, x4, loop
        add x7, x7, x5
        add x7, x7, x6
```

In the following five-stage pipelined RISC-V processor (IF, DEC, EXE, MEM, WB):

- All branches are predicted not-taken. (Always fetch from PC + 4).
- Branch decisions are made in the EXE stage.
- The pipeline has full bypassing.
- The processor annuls instructions following taken branches.
- Assume that in the first iteration of the loop both branches are taken.

(A) (label: 4A) (6 points) Her program just started running “start”. Fill in the pipeline diagram for the first 14 cycles. Show all bypass passed used in each cycle.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td>slli</td>
<td>add</td>
<td>lw</td>
<td>bge</td>
<td>mv</td>
<td>mv</td>
<td>mv</td>
<td>ori</td>
<td>addi</td>
<td>blt</td>
<td>add</td>
<td>add</td>
<td>add</td>
</tr>
<tr>
<td>DEC</td>
<td>addi</td>
<td>slli</td>
<td>add</td>
<td>lw</td>
<td>bge</td>
<td>bge</td>
<td>bge</td>
<td>mv</td>
<td>NOP</td>
<td>addi</td>
<td>blt</td>
<td>add</td>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>addi</td>
<td>slli</td>
<td>add</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>bge</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td>blt</td>
<td>NOP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>addi</td>
<td>slli</td>
<td>add</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>bge</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td>blt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>addi</td>
<td>slli</td>
<td>add</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>bge</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td>blt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(B) (2 points) How many cycles did it take to execute the first loop iteration on this processor? Make sure not to include the first two instructions at label \texttt{start} in your cycle count.

\textbf{(label: 4B) Cycles to execute first iteration of the loop on this processor: \underline{11}}

(C) (4 points) If you could modify your fetch stage to always fetch the correct next instruction instead of predicting all branches not taken, how many cycles will it now take to execute the first iteration of the loop on this modified processor? Explain your answer.

\textbf{(label: 4C) Removes both branch annulments, so saves 4 cycles, making the cycles per iteration \underline{11-4} = \underline{7}}.

(D) (4 points) Val spent so much money on vegan turkey that she couldn’t afford a processor with bypassing. In the cheapo processor she bought, all data hazards are resolved by stalling. Also, once again, all branches are predicted not taken.

Her program just started running “start”. \textbf{Fill in the pipeline diagram for the first 14 cycles:}

\textbf{(label: 4D)}

\begin{table}[h]
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
Cycle & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\hline
IF & addi & slli & add & add & add & lw & lw & lw & bge & bge & bge & bge & \\
\hline
DEC & addi & slli & slli & slli & slli & add & add & add & add & lw & lw & lw & lw & \\
\hline
EXE & addi & NOP & NOP & NOP & slli & NOP & NOP & NOP & add & NOP & NOP & NOP & NOP & \\
\hline
MEM & addi & NOP & NOP & NOP & NOP & NOP & NOP & NOP & add & NOP & NOP & NOP & \\
\hline
WB & addi & NOP & NOP & NOP & slli & NOP & NOP & NOP & NOP & NOP & NOP & add & NOP & \\
\hline
\end{tabular}
\end{table}

(E) (4 points) How many cycles did it take to execute the first loop on this processor? Hint: To answer this question use the bypass and stall information you determined in parts (A) and (D) to calculate how many cycles the first iteration of the loop would take on the cheapo processor. \textbf{Explain how you arrived at your solution.}

\textbf{(label: 4E_1) Cycles per iteration on this processor: \underline{24}}

\textbf{(label: 4E_2) Explanation: Without bypasses, there is a 3-cycle penalty for each EXE \rightarrow DEC bypass used in the bypassed pipeline, and a 1-cycle penalty for every WB \rightarrow DEC bypass used. In part B there were 4 EXE \rightarrow DEC bypasses and 1 WB \rightarrow DEC bypass, so cycles = 11 + 4*3 + 1*1 = 24.}
Extra pipeline diagrams (for parts A and D):

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>