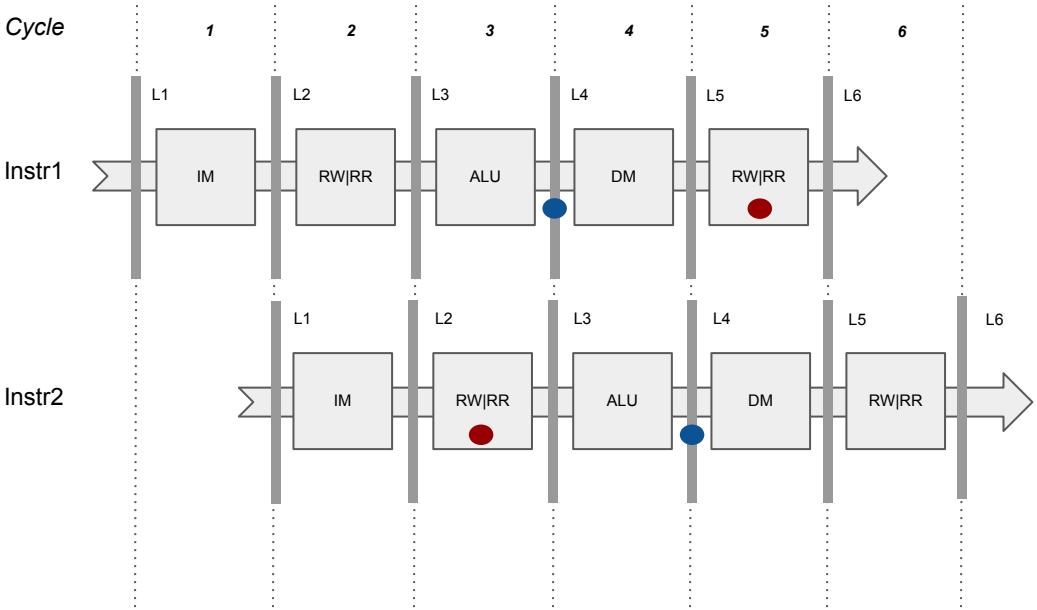
Using C, unrolling the loop 4 times:

```
for (i = 0 ; i< 25; i+=4) {
      a[i] = b[i]*c[i];
      a[i+1]=b[i+1]+c[i+1];
      a[i+2]=b[i+2]+c[i+2];
      a[i+3]=b[i+3]+c[i+3];
}
```

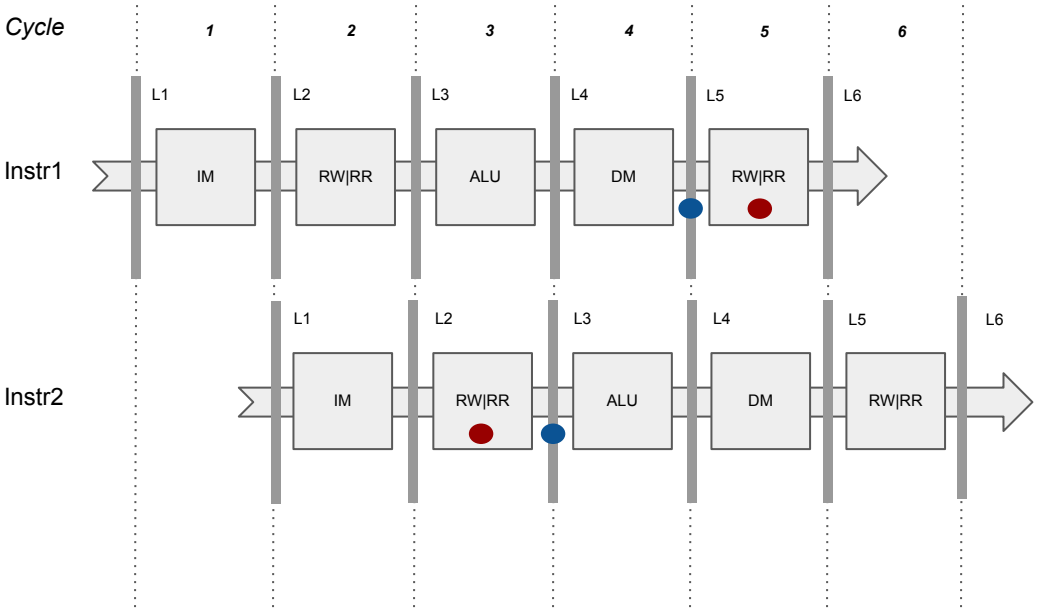


Part a) Integer addition followed by a dependent integer addition

- Example assembly code:
  Instr1: R1 <- R2 + R3
  Instr2: R4 <- R1 + R5

- *Finding number of stalls required without bypassing:*
  Instr1: R1 value is produced in cycle 5 (written on register).
  Instr2: R1 value is consumed in cycle 3 (read from register).
  Thus, we need two stalls.

- *Finding number of stalls required with bypassing (can use latch values):*
  Instr1: R1 value is available in Latch 4, end of cycle 3.
  Instr2: R1 value is used in beginning of cycle 4.
  Thus, we need zero stalls.

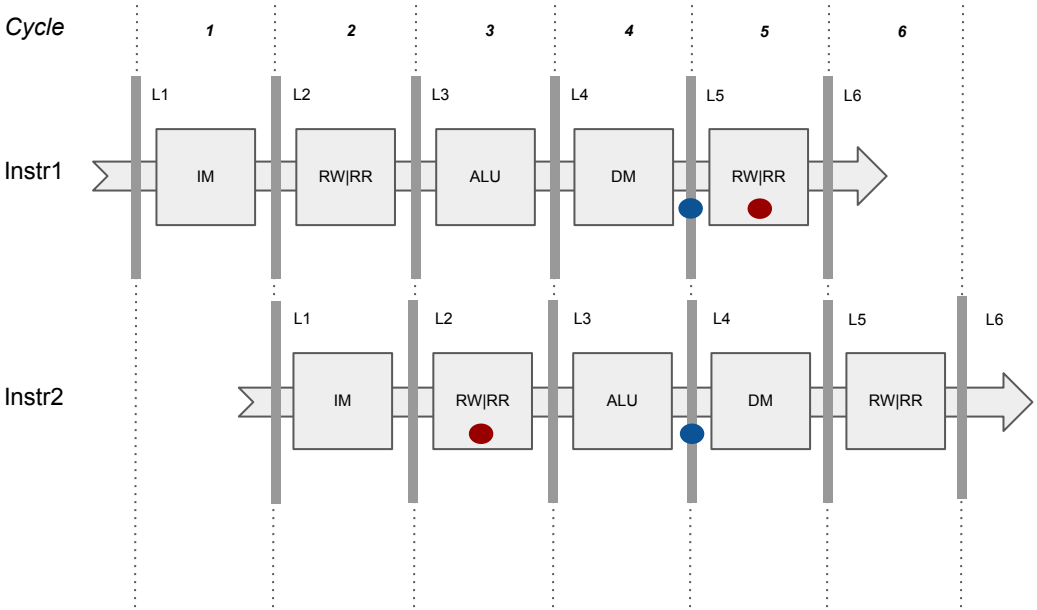


Part b) Load, providing the address for a store

- Example assembly code:
  Instr1: LD R1 <- [R2]
  Instr2: ST R3 -> [R1]

- *Finding number of stalls required without bypassing:*
  Instr1: R1 value is produced in cycle 5.
  Instr2: R1 value is consumed in cycle 3.
  Thus, we need two stalls.

- *Finding number of stalls required with bypassing (can use latch values):*
  Instr1: R1 value is available in Latch 5, end of cycle 4.
  Instr2: R1 value is used to store it in address in R3 in beginning of cycle 5.
  Thus, we need 1 stall.

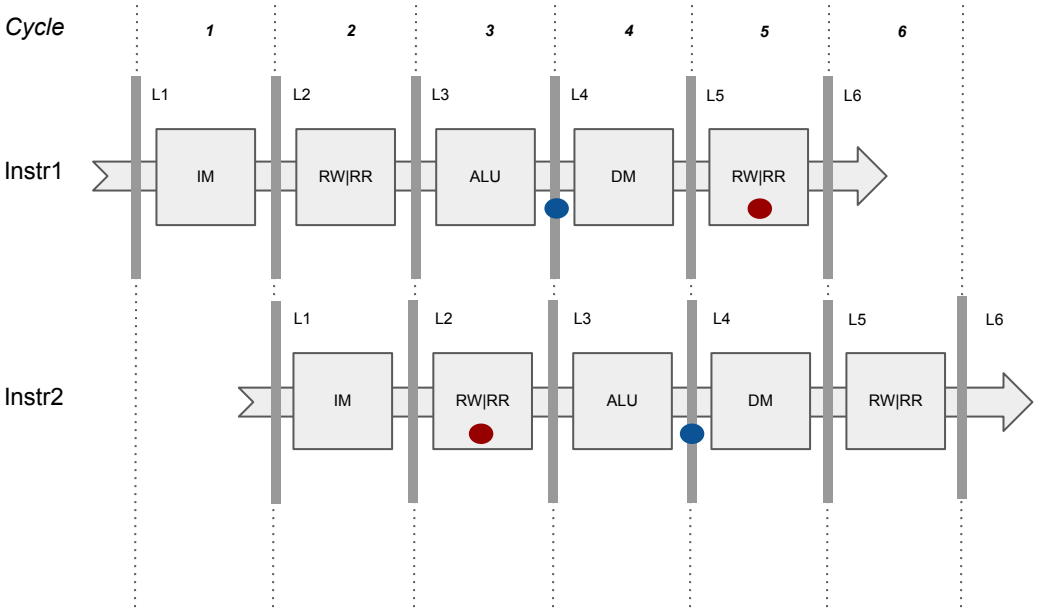


Part c) Load, providing the data for a store

- Example assembly code:
  Instr1: LD R1 <- [R2]
  Instr2: ST R1 -> [R3]

- *Finding number of stalls required without bypassing:*
  Instr1: R1 value is produced in cycle 5.
  Instr2: R1 value is consumed in cycle 3.
  Thus, we need two stalls.

- *Finding number of stalls required with bypassing (can use latch values):*
  Instr1: R1 value is available in Latch 5, end of cycle 4.
  Instr2: R1 value is used to store it in address in R3 in beginning of cycle 5.
  Thus, we need zero stalls.



Part d) Integer addition providing the address for the store

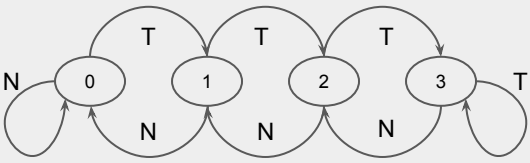
- Example assembly code:
  Instr1: R1 <- R2 + R3
  Instr2: ST R3 -> [R1]

- *Finding number of stalls required without bypassing:*
  Instr1: R1 value is produced in cycle 5.
  Instr2: R1 value is consumed in cycle 3.
  Thus, we need two stalls.

- *Finding number of stalls required with bypassing (can use latch values)*:
  Instr1: R1 value is available in Latch 4, end of cycle 3.
  Instr2: R1 value is used in beginning of cycle 5.
  Thus, we need zero stalls.

Part a)

2-bit counter states:



N - Branch not taken
T - Branch taken
0 - Strongly not taken
1 - Weakly not taken
2 - Weakly taken
3 - Strongly taken

```
int a [ ] = { 1, 0, 1, 0, 0 };

while (1) {
        for (i = 0; i< 5; i++) {//BR1
                if (a[i]==0) {    //BR2
                        …
                }
        }
}
```

- ❏ The above program has 2 branches BR1 and BR2 (*highlighted*).
- ❏ Each branch has a 2-bit counter.
- ❏ Counters are initialized to 0.
- ❏ First get steady state values of each counter, by running the program for a large number of iterations (a pattern will appear).
- ❏ From a quick run down on the execution table (*right*), we see steady values of the BR1's and BR2's counters at the beginning of each set of for loop iterations (i.e. the number of times the for loop branch executes *in a single while loop*) are 2 and 3 respectively (or 10 and 11 respectively in binary.)
- ❏ At steady state, let's now run our for loop for another set of iterations, with steady state values of 2 and 3 for BR1 and BR2 respectively, we get:

	For Loop Iteratn. #	[BR1 Taken/Not Taken, BR 1 Counter value]
for loop iteration set 1	1	[T,1]
	2	[T,2]
	3	[T,3]
	4	[T,3]
	5	[T,3]
	6	[N,2]
for loop iteration set 2	7	[T,3]
	8	[T,3]
	9	[T,3]
	10	[T,3]
	11	[T,3]
	12	[N,2]
for loop iteration set 3	13	[T,3]
	14	[T,3]
	15	[T,3]
	16	[T,3]
	17	[T,3]
	18	[N,2]
	…	…

For Loop Iteratn. #	[BR1 Taken/Not Taken, BR 1 Counter value] after iteration	BR1 Prediction	[BR2 Taken/Not Taken, BR 2 Counter value] after iteration	BR2 Prediction
1	[T,3]	correct	[N,2]	mispredict
2	[T,3]	correct	[T,3]	correct
3	[T,3]	correct	[N,2]	mispredict
4	[T,3]	correct	[T,3]	correct
5	[T,3]	correct	[T,3]	correct
6	[N,2]	mispredict	[-,3]	-

- ❏ 11 branches were executed, with 8 correct predictions and 3 mispredictions. So success rate out of 10 branches is (8/11)*10 = 7.27.

Part b)

- ❏ Explain local predictor. (Rajeev's slides)

References:
http://www.cs.utah.edu/~rajeev/cs6810/
https://www.quora.com/CPUs-How-is-branch-prediction-implemented-in-microprocessors