# ECE 5643
# Advanced Computer Architecture
# Midterm Exam

May 4, 2023

Name:____________________

No collaboration of any kind is allowed!

| Problem | Points |
|---|---|
| Problem 1 | / 30 |
| Problem 2 | / 30 |
| Problem 3 | / 20 |
| Problem 4 | / 20 |
| Total | / 100 |

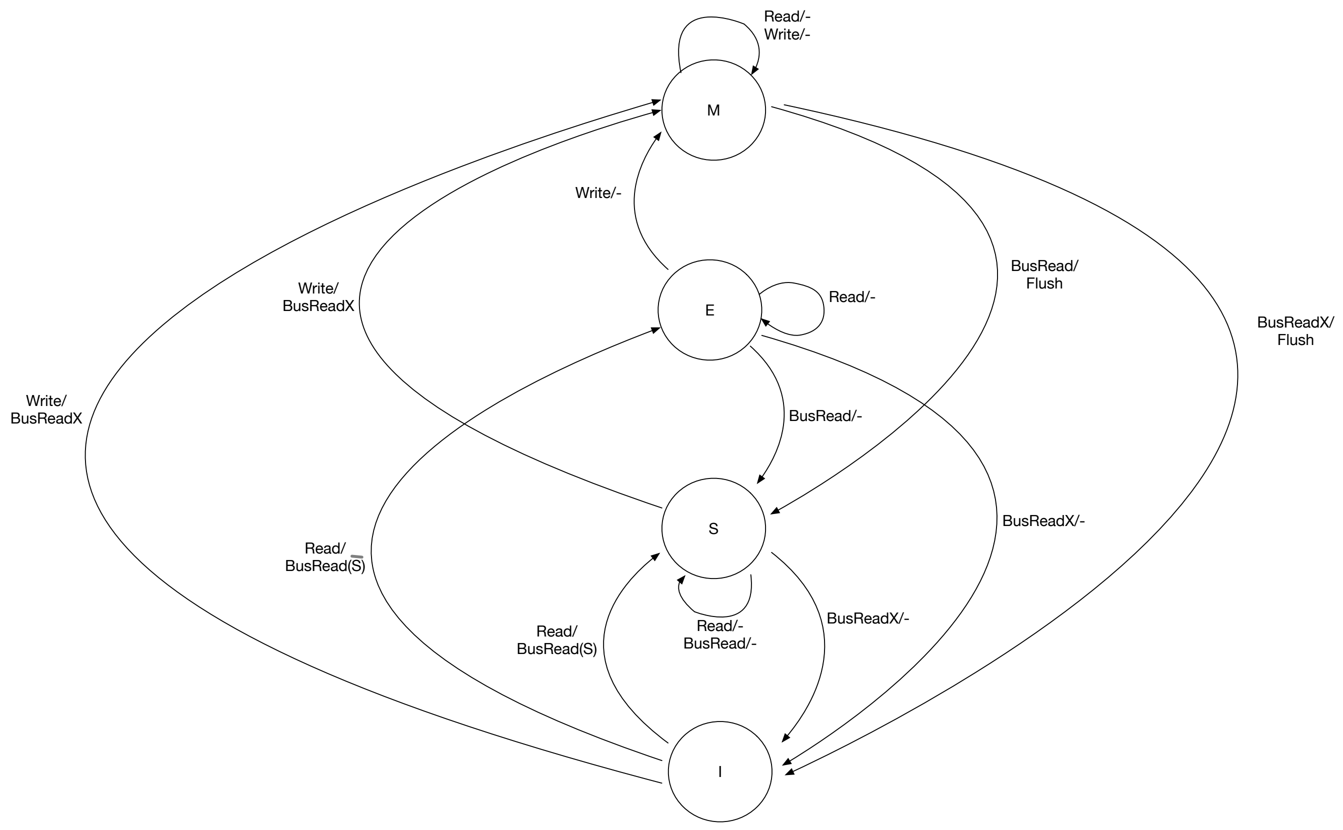
On all problems, please show your work for any chance at partial credit!

Problem 1) 40 points In this problem we compare an invalidation based protocol with an update protocol.

i. Explain in one sentence the difference between an invalidation protocol and an update protocol.

In an invalidate protocol, when a processor writes a block, that results in all other processors *invalidating* that same block in their caches, whereas in an update protocol a processor that writes a block will instead *update* all other processors that have that same block in their caches.

The following is the MESI invalidation protocol.



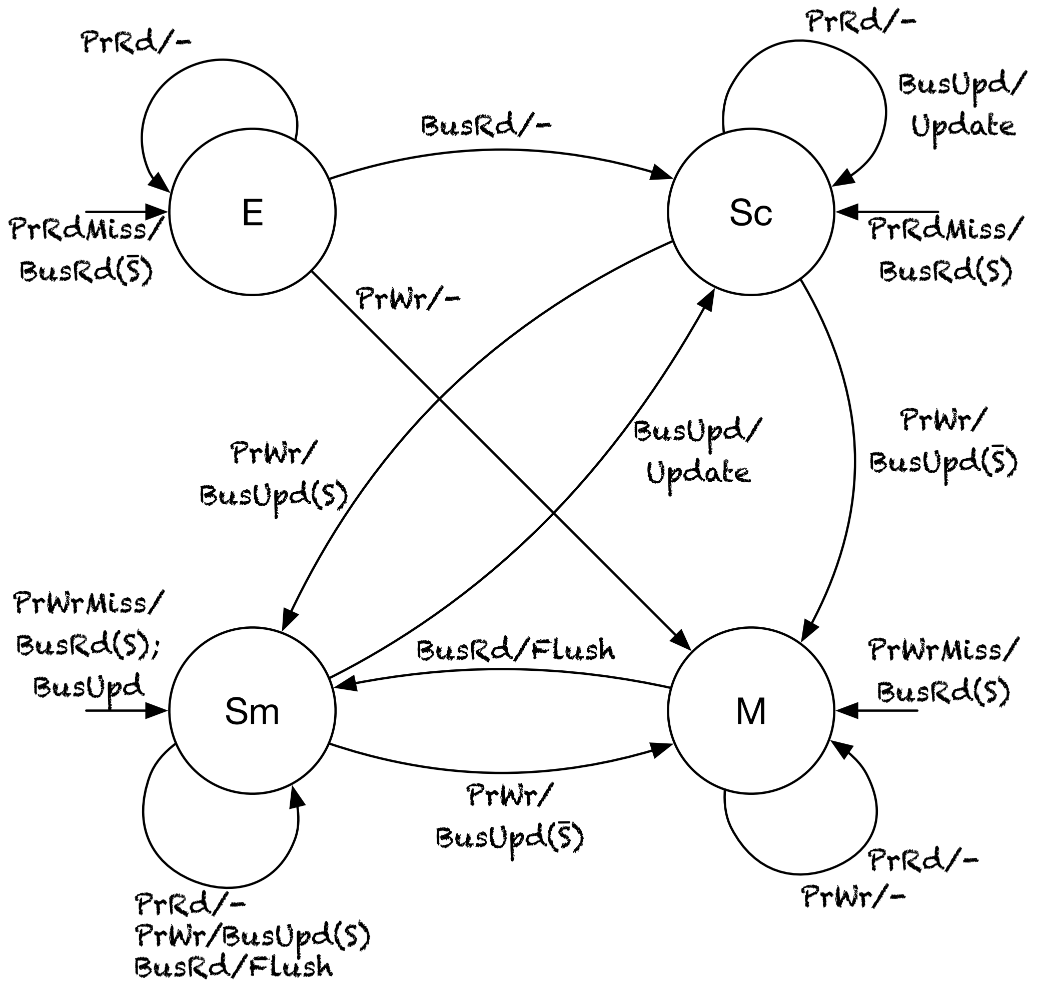
ii. Explain the benefit of the E stage in the MESI protocol.

The Exclusive state is entered whenever a processor reads a block and no other processors have that block int their caches. When that same processor then writes that block, it can enter the Modified state without performing any bus transaction (because no other processor can have that block in its cache).

iii. For the following sequence of reads and writes to a shared memory location, show the state of the shared block in each processors cache using the MESI protocol. Show also any bus transactions that occur for that memory access.

| | P1 | P2 | P3 | Bus Transactions |
|---|---|---|---|---|
| | I | I | I | |
| P1 reads | E | I | I | BusRead(S) |
| P1 writes | M | I | I | - |
| P2 reads | S | S | I | P2: BusRead(S)<br>P1: Flush |
| P1 reads | S | S | I | - |
| P3 reads | S | S | S | BusRead(S) |
| P2 writes | I | M | I | BusReadX |
| P1 writes | M | I | I | P1: BusReadX<br>P2: Flush |
| P2 reads | S | S | I | P2: BusRead(S)<br>P1: Flush |
| P3 writes | I | I | M | BusReadX |

The following is the Dragon update protocol.



iv. For the following sequence of reads and writes to a shared memory location, show the state of the shared block in each processors cache using the Dragon protocol. Show also any bus transactions that occur for that memory access.

| | P1 | P2 | P3 | Bus Transactions |
|---|---|---|---|---|
| | I | I | I | |
| P1 reads | E | I | I | BusRead(S) |
| P1 writes | M | I | I | - |
| P2 reads | Sm | Sc | I | P2: BusRead(S)<br>P1: Flush |
| P1 reads | Sm | Sc | I | - |
| P3 reads | Sm | Sc | Sc | P3: BusRead(S)<br>P1: Flush |
| P2 writes | Sc | Sm | Sc | P2: BusUpd(S)<br>P1: Update<br>P3: Update |
| P1 writes | Sm | Sc | Sc | P1: BusUpd(S)<br>P2: Update<br>P3: Update |
| P2 reads | Sm | Sc | Sc | - |
| P3 writes | Sc | Sc | Sm | P3: BusUpd(S)<br>P1: Update<br>P2: Update |

v. Assume that the performance of the code that generated the sequence of reads and writes in parts iii. and iv. is completely bound by the performance of the memory bus. Assume also that each different type of memory transaction takes the following # of cycles:

| | |
|---|---|
| BusRd | 2 |
| BusRdX | 2 |
| Flush | 5 |
| BusUpd | 3 |
| Update | 0 |

How many bus cycles does this program take on systems using MESI or Dragon coherence approaches?

MESI: 4 BusRd, 3 BusRdX, 3 Flush

4 * 2 + 3 * 2 + 3 * 5 = 8 + 6 + 15 = <u>29 cycles</u>

Dragon: 3 BusRd, 2 Flush, 3 BusUpd (and Updates take 0 cycles)

3 * 2 + 2 * 5 + 3 * 3 = 6 + 10 + 9 = <u>25 cycles</u>

Problem 2) 40 points
The following problems relate to consistency.

i) Of the consistency models we've studied, which model is most likely to match a programmer's intuition?

Sequential consistency

ii) Why don't all multiprocessor systems implement the intuitive consistency model in i)?

Sequential consistency is the most expensive consistency model to implement and because it is the most restrictive model, it has lower performance than any relaxed model.

Assume two shared memory locations, X and Y are initially 0.

| Processor 1 | Processor 2 |
|---|---|
| X = 1 | Y = 1 |
| r1 = Y | r2 = X |

iv. Assuming **sequential consistency**, give all the possible values for registers in the following form:
(r1, r2)

(0, 1)
(1, 0)
(1, 1)

iv. Assuming **total store ordering consistency**, give all the possible values for registers in the following form:
(r1, r2)

(0, 1)
(1, 0)
(1, 1)

v. What is the benefit of the more relaxed **total store ordering consistency** compared to **sequential consistency**?

Total store ordering allows a store that is followed by an independent load to be reordered with that load. One benefit of this is that stores can be buffered before they are written out to memory and won't delay loads that follow from accessing memory.

| Processor 1 | Processor 2 |
|---|---|
| A = 1 | print A, B |
| B = 1 | |

vi. What are the possible values printed under each of the following consistency models:

Sequential consistency - 0, 0    1, 0    1, 1

Total store ordering consistency -
Doesn't allow any additional reordering so same as SC: 0, 0    1, 0    1, 1

Partial store ordering consistency -
Allows the stores to A and B to be reordered so 0, 1 is possible as well as all possible SC results: 0, 0    0, 1    1, 0    1, 1

Weak consistency -
In this case, PSO allows all possible outputs so even though Weak is a more relaxed model, the possible results under the weak model is the same:
0, 0    0, 1    1, 0    1, 1

vii. Assume shared memory locations A and B are initially 0:

| Processor 1 | Processor 2 | Processor 3 |
|---|---|---|
| A = 1 | u = A | v = B |
| | B = 1 | w = A |

Assuming **sequential consistency**, show all the possible values for u, v and w after these three threads execute:

u = 0, v = 0, w = 0
u = 0, v = 0, w = 1
u = 0, v = 1, w = 0
u = 0, v = 1, w = 1
u = 1, v = 0, w = 0

u = 1, v = 0, w = 1
u = 1, v = 1, w = 1

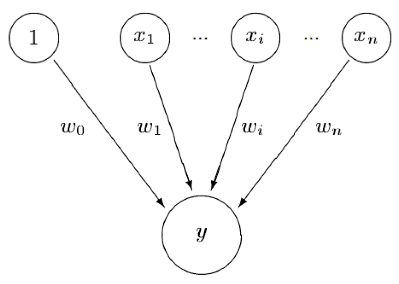
( u = 1, v = 1, w = 0 is the only combination that is not possible under SC )

vii. Circle which of the following relaxed consistency models could have a different result than that of sequential consistency in vi):

Only weak consistency allows u = 1, v = 1, w = 0 as a possible result.

Problem 3 (20 points). Assume the following branch is predicted by a perceptron predictor such as the one discussed in class and in the paper on Canvas.

BGT R1, 3, TARG ; branch to TARG if R1 > 3



$$y = w_0 + \sum_{i=1}^{n} x_i w_i$$

```
for each bit in parallel
    if $t = x_i$ then
        $w_i := w_i + 1$
    else
        $w_i := w_i - 1$
    end if
```

Perceptron Model (from Jimenez and Lin)

For the following values of R1, show in a table, the values of the weights when the predication is made, the output Y, the prediction (Taken vs. Not taken) and the actual outcome of the branch. Assume four bits of history are used. Start with a history of 0000.

| R1 | w0 | w1 | w2 | w3 | w4 | y | Prediction | Actual Outcome |
|---|---|---|---|---|---|---|---|---|
| 0 | -2 | -1 | 1 | 0 | 2 | -4 | NT | NT |

Bias input x0 is always 1, x1 = -1 x2 = -1 x3 = -1, x4 = -1
y = 1 * -2 + -1 * -1 + -1 * 1 + -1 * 0 + -1 * 2 = -2 + 1 + -1 + 0 + -2 = -4 => prediction NT
w0 decremented, w1 incremented, w2 incremented, w3 incremented, w4 incremented
Actual outcome NT is shifted in as new x4 (-1).

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 3 | -3 | 0 | 2 | 1 | 3 | -9 | NT | NT |

Bias input x0 is always 1, x1 = -1 x2 = -1 x3 = -1, x4 = -1
y = 1 * -3 + -1 * 0 + -1 * 2 + -1 * 1 + -1 * 3 = -3 + 0 + -2 + -1 + -3 = -9 => prediction NT
w0 decremented, w1 incremented, w2 incremented, w3 incremented, w4 incremented
Actual outcome NT is shifted in as new x4 (-1).

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 6 | -4 | 1 | 3 | 2 | 4 | -14 | NT | T |

Bias input x0 is always 1, x1 = -1 x2 = -1 x3 = -1, x4 = -1
y = 1 * -4 + -1 * 1 + -1 * 3 + -1 * 2 + -1 * 4 = -4 + -1 + -3 + -2 + -4 = -14 => prediction NT
w0 incremented, w1 decremented, w2 decremented, w3 decremented, w4 decremented
Actual outcome T is shifted in as new x4 (1).

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | -3 | 0 | 2 | 1 | 3 | -3 | NT | NT |

Bias input x0 is always 1, x1 = -1 x2 = -1 x3 = -1, x4 = 1
y = 1 * -3 + -1 * 0 + -1 * 2 + -1 * 1 + 1 * 3 = -3 + 0 + -2 + -1 + 3 = -3 => prediction NT
w0 decremented, w1 incremented, w2 incremented, w3 incremented, w4 decremented
Actual outcome NT is shifted in as new x4 (-1).

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2 | -4 | 1 | 3 | 2 | 2 | -8 | NT | NT |

Bias input x0 is always 1, x1 = -1 x2 = -1 x3 = 1, x4 = -1
y = 1 * -4 + -1 * 1 + -1 * 3 + 1 * 2 + -1 * 2 = -4 + -1 + -3 + 2 + -2 = -8 => prediction NT
w0 decremented, w1 incremented, w2 incremented, w3 decremented, w4 incremented
Actual outcome NT is shifted in as new x4 (-1).

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | -5 | 2 | 4 | 1 | 3 | -7 | NT | T |

Bias input x0 is always 1, x1 = -1 x2 = 1 x3 = -1, x4 = -1
y = 1 * -5 + -1 * 2 + 1 * 4 + -1 * 1 + -1 * 3 = -5 + -2 + 4 + -1 + -3 = -7 => prediction NT
w0 incremented, w1 decremented, w2 incremented, w3 decremented, w4 decremented
Actual outcome T is shifted in as new x4 (1).

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | -4 | 1 | 5 | 0 | 2 | -6 | NT | T |

Bias input x0 is always 1, x1 = 1 x2 = -1 x3 = -1, x4 = 1
y = 1 * -4 + 1 * 1 + -1 * 5 + -1 * 0 + 1 * 2 = -4 + 1 + -5 + 0 + 2 = -6 => prediction NT
w0 incremented, w1 incremented, w2 decremented, w3 decremented, w4 incremented
Actual outcome T is shifted in as new x4 (1).

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | -3 | 2 | 4 | -1 | 3 | -7 | NT | T |

Bias input x0 is always 1, x1 = -1 x2 = -1 x3 = 1, x4 = 1
y = 1 * -3 + -1 * 2 + -1 * 4 + 1 * -1 + 1 * 3 = -3 + -2 + -4 + -1 + 3 = -7 => prediction NT
w0 incremented, w1 decremented, w2 decremented, w3 incremented, w4 incremented
Actual outcome T is shifted in as new x4 (1).

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | -2 | 1 | 3 | 0 | 4 | 4 | T | NT |

Bias input x0 is always 1, x1 = -1 x2 = 1 x3 = 1, x4 = 1
y = 1 * -2 + -1 * 1 + 1 * 3 + 1 * 0 + 1 * 4 = -2 + -1 + 3 + 0 + 4 = 4 => prediction T
w0 decremented, w1 incremented, w2 decremented, w3 decremented, w4 decremented
Actual outcome NT is shifted in as new x4 (-1).

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | -3 | 2 | 2 | -1 | 3 | -3 | NT | T |

Bias input x0 is always 1, x1 = 1 x2 = 1 x3 = 1, x4 = -1
y = 1 * -3 + 1 * 2 + 1 * 2 + 1 * -1 + -1 * 3 = -3 + 2 + 2 + -1 + -3 = -3 => prediction NT
w0 incremented, w1 incremented, w2 incremented, w3 incremented, w4 decremented
Actual outcome T is shifted in as new x4 (1).

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 6 | -2 | 3 | 3 | 0 | 2 | 6 | T | T |

Bias input x0 is always 1, x1 = 1 x2 = 1 x3 = -1, x4 = 1
y = 1 * -2 + 1 * 3 + 1 * 3 + -1 * 0 + 1 * 2 = -2 + 3 + 3 + 0 + 2 = 6 => prediction T

Problem 4 (20 Points)

Consider the following CPI breakdown for a multicycle processor implementation running an application of interest:

| Instruction | Percentage | Cycles |
|---|---|---|
| Integer ALU | 30% | 1 |
| Load/Store | 30% | 2 |
| Shift | 10% | 3 |
| Floating Point | 10% | 5 |
| Branches | 20% | 1 |

a) Compute the overall CPI for a multicycle processor implementation assuming **perfect** branch prediction.

.3 * 1 + .3 * 2 + .1 * 3 + .1 * 5 + .2 * 1 = .3 + .6 + .3 + .5 + .2 = 1.9 cycles per instruction

b) Compute the overall CPI assuming 95% of branches are predicted correctly. Assume a 5 cycle penalty for mispredicted branches.

Note, this answer assumes the 5 cycle penalty is *in addition* to the 1 cycle execution of all branches:
.3 * 1 + .3 * 2 + .1 * 3 + .1 * 5 + .2 * (.95 * 1 + 0.05 * 6) = .3 + .6 + .3 + .5 + .19 + 0.06 = 1.95 cycles per instruction

c) What is the speedup of perfect branch prediction over the 95% branch prediction of part b)?

$$\begin{aligned}\text{Speedup}_{\text{perfect BP/95\% BP}} &= \text{Performance}_{\text{perfect BP}} / \text{Performance}_{\text{95\% BP}} \\ &= \text{Execution Time}_{\text{95\% BP}} / \text{Performance}_{\text{perfect BP}} \\ &= 1.95 / 1.9 = \underline{1.026X}\end{aligned}$$

Now consider the an out-of-order processor:

The latencies for different types of instructions are given in the table for each instruction. The latencies in the table as such that if an instruction with a latency of 1 issues in cycle i, a consumer can issue in cycle i+1.

| **opcode** | **latency** |
|---|---|
| add | 1 |
| load | 2 |
| mul | 3 |
| branch | 1 |
| store | 1 |

The Reservation Stations for the out-of-order processor are shown. Notice that architectural register names have already been reordered to physical register names. Assume the instructions are numbered in program order (i.e. 101 comes before 102, etc.).

| | **Src1** | **Src2** | **Dest** | |
|---|---|---|---|---|
| 101 load | - | - | PR6 | XY |
| 102 mul | PR6 | - | PR7 | XY |
| 103 branch | PR6 | - | - | XY |
| 105 store | - | PR7 | - | |
| 106 add | - | - | PR11 | XY |
| 107 add | PR11 | PR6 | PR12 | X |
| 109 add | PR12 | - | PR14 | X |
| 110 add | - | - | PR15 | X |
| 111 load | PR15 | - | PR16 | |
| 112 store | - | PR16 | - | |

Reservation Stations

a) Which instructions are initially woken up (i.e. ready to be issued out of the RS)?

101, 106, 110 are not waiting on any sources and are thus all woken up

b) For the instructions in the Reservation Stations, show in which cycle each instruction would issue assuming the latencies above.

| Cycle Issued | Instruction | Cycle completed |
|---|---|---|
| 1 | 101 load | 3 |
| 2 | 106 add | 3 |
| 3 | 102 mul | 6 |
| 4 | 103 branch | 5 |
| 5 | 107 add | 6 |
| 6 | 105 store | 7 |
| 7 | 109 add | 8 |
| 8 | 110 add | 9 |
| 9 | 111 load | 11 |
| 10 | - | - |
| 11 | 112 store | 12 |

c) Assume that instructions are numbered in program order. Explain why instruction 104 and 108 are not found in the Reservation Stations?

They are no longer in the reservation stations because they have already completed their execution

| INST | 112 ST | 111 LD | 110 add | 109 add | 108 BR | 107 add | 106 add | 105 ST | 104 ADD | 103 BR | 102 mul | 101 LD | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Phys Reg | PR17 | PR16 | PR15 | PR14 | PR13 | PR12 | PR11 | PR10 | PR9 | PR8 | PR7 | PR6 | |
| Com-plete ? | | | | | X | | | | X | | | | |

ROB

d) Explain briefly the need for the ROB. Use the instructions in the example above in your answer as much as possible in your answer.

The Reorder Buffer holds the result of executed instructions so that they can be committed to architectural registers in the original program order. This is needed in the case of branch prediction misses (or exceptions) so that instructions that would never have executed in an in-order processor (i.e. a processor that executes instructions in the original program order). For example, if the branch instruction 103 is a mispredicted branch, the following instruction, 104, should not have executed. When that instruction 104 completed its execution, it wrote the physical register that is part of the reorder buffer rather than the architectural register destination of instruction 104. Since 104's result has not been committed, instructions 104-112 can be flushed out of the processor. 104's result would only be committed to its architectural register destination if 104 was the oldest instruction in the reorder buffer.

e) Assume the program in the Reservation Stations and ROB in the examples above. In the table below, show in which cycle each instruction in the ROB would be committed. Complete the table ONLY up to cycle 12. For any cycle in which no instructions are committed put a dash (—). Assume instructions in Reservation Stations are complete once they have been issued and their latency have passed. For example, if an instruction is issued in cycle 1 and has a latency of 1, it executes in cycle 2 observing its 1-cycle latency and is complete in cycle 3. Assume multiple instructions can be committed in a single cycle.

| Cycle | Instruction | | | |
|---|---|---|---|---|
| 4 | 101 LD | | | |
| 7 | 102 MUL | 103 BRANCH | 104 ADD | |
| 8 | 105 ST | 106 ADD | 107 ADD | 108 BRANCH |
| 9 | 109 ADD | | | |
| 10 | 110 ADD | | | |
| 11 | - | | | |
| 12 | 111 LD | | | |