CS224
Fall 2020
QUIZ NO. ~~5~~ 4 SOLUTIONS

December 5, 2020

PLEASE READ: 1. Only handwritten answers are accepted. 2. Convert your handwritten answers to pdf and upload one pdf file to Moodle. Make sure that you answer the questions in the order they are given. Provide a neat work and make sure that your answers are numbered and in the order of questions and distinguishable from each other. Do not miss the deadline.

1. In this question consider the branch prediction problem. Consider a new branch prediction algorithm with 6 states. The new states in addition to the ones we have in 2-bit prediction algorithm are *Predict Very Strongly Taken* and *Predict Very Strongly Not Taken*. If we are in Predict Strongly Taken state and a branch is taken, it goes to the *Predict Very Strongly Taken* state and it stays there if the next branch is taken. If we are in the *Predict Very Strongly Taken* state and a branch is not taken it goes to Predict Strongly Taken state. A similar type of state transition is applied to *Predict Very Strongly Not Taken* state.

i. Draw the state transition diagram for the new branch prediction algorithm.

ii. Consider the following for a beq instruction: T N T T N N N N N T
The above sequence shows that branch is taken while executing the beq instruction during the first execution. In the second execution it is not taken, etc. Use the new branch prediction algorithm with six states for branch prediction. Assume that the initial state is Branch Strongly Taken.

Show the success of the algorithm in each execution of the instruction. Use + for correct prediction and use - for incorrect prediction. What is the prediction success rate of the algorithm?

State: BST BVST BST BVST BVST BST BWT BWNT BSNT BVSNT BSNT ← final state

Current state ↓ Actual Branch ↗ Next state

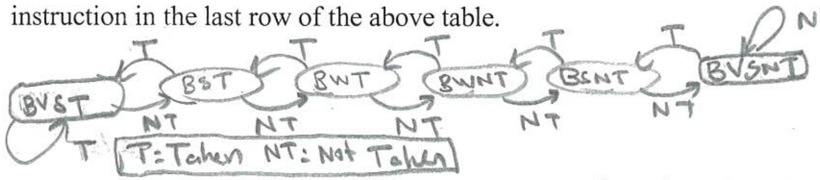
Inst. No.	1	2	3	4	5	6	7	8	9	10
Branch Taken or Not Taken	T	N	T	T	N	N	N	N	N	T
Algorithm Decision	+	–	+	+	–	–	–	+	+	–
No. of Clock Cycles	1	5	1	1	5	5	5	1	1	5

← +: Correct decision
–: Incorrect decision

Success Rate: 5/10 ⇒ 50 %

iii. Consider the use of the new branch prediction algorithm in a 8 stage pipeline. Assume that the branch decision is made in the 5th stage of the pipeline. Give the number of clock cycles required for each instruction in the last row of the above table.

i.



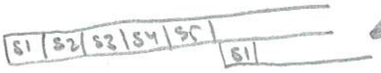
Branch very strongly taken | Branch strongly taken | Branch weakly taken | Branch weakly Not taken | Branch strongly not taken | Branch very strongly not taken

Calculate CPI for these 10 instructions

CPI = (Total no. of clock cycles) / (No. of instructions)

$= \frac{30}{10} = 3$

S1: IF (Stage 1)



← if branch prediction is incorrect branch instruction requires 5 clock cycles, otherwise 1 clock cycle

predict = true

Total of 30 clock cycles.

CS 224
Fall '20
Quiz No. 4
Sol. (cont.)

No cycle is lost in branch instructions

2. Consider a simple 3-stage pipeline datapath implementation, in which there is a Fetch/Decode stage, followed by an Execute stage, followed by a Memory/WriteBack stage. The stages are separated by pipeline registers, and there is hardware in the Fetch/Decode stage to determine jumps and branches. But there is no hardware to support flushes, stalls, forwarding, or early write-late read (i.e. all clock inputs are positive edge triggered). This "barely pipelined" processor must run the following code correctly, without errors:

RAW: read after write

F/D / EX / MWB

```
      ...
Top:① lw   $a0, 0($s0)        # Get next array item
    ② add  $s1, $s1, $a0            # Add it into the total
    ③ sub  $t2, $s0, $t1            # Calculate the store address
    ④ sw   $s1, 0($t2)        # Store the partial total
    ⑤ addi $s0, $s0, 4              # Calculate address of next array item
    ⑥ bne  $s0, $v1, Top            # If not at final address, do the next item
      ...
```

2 nops are needed for $a0.

2 nops for $t2

2 nops for $s0

There are dependencies between instructions:
1-2, 2-4, 5-6

1-2:
(*) IF/ID | EX | MWB
IF

2-4:

5-6:

(*) Between 1-2 two instructions are needed.

i. By simply adding enough *nops*, hazards can be eliminated. Without changing the order of the instructions, give the hazard free code, showing in the code above where the *nops* should be inserted, and how many (use the minimum number of *nop* instructions). Mark the insertion points for the *nop* instructions in the code above and indicate how many are needed at each place and which register(s) they are needed for—DO NOT rewrite the code again. For the modified program with your *nops* inserted so that the code executes correctly on the 3-stage MIPS pipeline processor with no forwarding or stalling or flush hardware, how many clock cycles are needed to process a 100-item array? Each iteration 12 cycles 100×12=1200 clock cycles

ii. Again consider the same code segment repeated below.

```
Top:  lw   $a0, 0($s0)      # Get next array item
      add  $s1, $s1, $a0          # Add it into the total
      sub  $t2, $s0, $t1          # Calculate the store address
      sw   $s1, 0($t2)      # Store the partial total
      addi $s0, $s0, 4            # Calculate address of next array item
      bne  $s0, $v1, Top          # If not at final address, do the next item
```

By rearranging instruction order, hazards can sometimes be eliminated or their effect reduced. In this part, you may reorder the code and add *nops* (as a compiler would) in order to make it hazard free and faster. Give the reodered code as specified for the following cases and mark the insertion points for the *nop* instructions in the code above and indicate how many are needed at each place and which register(s) they are needed for. For the modified program with your *nops* inserted so that the code executes correctly on the 3-stage MIPS pipeline processor with no forwarding or stalling or flush hardware, how many clock cycles are needed to process a 100-item array in each case?

Case 1: Reorder instructions such that such that there will be only three nops.

Case 2: Reorder instructions such that there will be only two nops.

Case 1:

```
Top: lw $a0, 0($s0)
  # nop
     sub $t2, $s0, $t1
     add $s1, $s1, $a0
  # nop for $s1
     addi $s0, $s0, 4
     sw $s1, 0($t2)
  # nop for $s0
     bne $a0, $v1, Top
```

Case 2:

```
Top: lw $a0, 0($s0)
     sub $t2, $s0($t1)
     addi $s0, $s0, 4
     add $s1, $s1, $a0
  # nop  nop (2 nops)
     sw $s1, 0($t2)
     bne $s0, $v1, Top
```