# Computer Architecture LSU EE 4720 <br> Final Examination 

Monday, 9 May 2022 10:00-12:00 CDT

Problem $1 \longrightarrow(20 \mathrm{pts})$
Problem $2 \longrightarrow(20 \mathrm{pts})$
Problem $3 \longrightarrow(20 \mathrm{pts})$
Problem $4 \longrightarrow(20 \mathrm{pts})$
Problem $5 \longrightarrow(20 \mathrm{pts})$
Alias Purple Mode.
Exam Total $\quad(100 \mathrm{pts})$

Problem 1: (20 pts) Show the execution of the code fragments on the following implementations for enough iterations to determine the instruction throughput (IPC). As always, base the behavior of branches and the availability of bypasses on the implementations. Also, don't forget that MIPS branches have a delay slot. Sorry for yelling, but I hate it when students miss things.


Show execution and $\checkmark$ determine instruction throughput (IPC) based on a large number of iterations.
 is in WB. Two wrong-path instructions are fetched, xor and sub. They are squashed when the branch is resolved. (Of course, they would not be squashed if the branch were not taken.)
The instruction throughput is $\frac{2 \mathrm{insn}}{(8-4) \text { cyc }}=\frac{2}{4}$ insn/cycle based on the second iteration starting at cycle 4 and the third iteration starting at cycle 8.

```
    # SOLUTION -- Dynamic Instruction Order
LOOP: # Cycle 0
    bne r1, r2, LOOP IF ID EX ME WB # First Iteration
    addi r1, r1, 4 IF ID EX ME WB
    xor r5, r6, r7 IF IDx
    sub r8, r9, r10 IFx
LOOP: # Cycle }0
    bne r1, r2, LOOP IF ID EX ME WB # Second Iteration
    addi r1, r1, 4 IF ID EX ME WB
    xor r5, r6, r7 IF IDx
    sub r8, r9, r10 IFx
LOOP: # Cycle }0
    bne r1, r2, LOOP IF ID EX ME WB
```

```
    # These instructions will be completely executed after the last iteration.
```

    # These instructions will be completely executed after the last iteration.
    xor r5, r6, r7
    xor r5, r6, r7
    sub r8, r9, r10
    ```
    sub r8, r9, r10
```



Show execution and $\checkmark$ determine instruction throughput (IPC) based on a large number of iterations.
The solution appears below. The good news in this pipeline the branch is resolved in ID, meaning that zero wrong-path instructions are fetched. The bad news is that there is a dependence carried by r 1 that stalls bne in ID for two cycles. For this reason, the instruction throughput is the same: $\frac{2 \text { insn }}{(6-2) \mathrm{cyc}}=\frac{2}{4}$ insn/cycle based on the second iteration starting at cycle 2 and the third iteration starting at cycle 6.

```
LOOP: # Code in Static Instruction Order
    bne r1, r2, LOOP
    addi r1, r1, 4
    xor r5, r6, r7
    sub r8, r9, r10
```

    \# SOLUTION -- Dynamic Instruction Order
    LOOP: \# Cycle $\quad 0 \quad 1 \quad 2 \quad 3 x_{1}$
bne r1, r2, LOOP IF ID EX ME WB \# First Iteration
addi r1, r1, 4 IF ID EX ME WB

bne r1, r2, LOOP IF ID ----> EX ME WB \# Second Iteration
addi r1, r1, 4 IF ----> ID EX ME WB
LOOP: \# Cycle $\quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5$
bne r1, r2, LOOP IF ID ----> EX ME WB
addi r1, r1, 4 IF ----> ID EX ME WB
\# These instructions will be executed after the last iteration.
xor r5, r6, r7
sub r8, r9, r10


Show execution and $\nabla$ determine instruction throughput (IPC) based on a large number of iterations.
In this implementation there is a bypass that helps with the branch condition dependence, reducing the stall from two cycles to one cycle. The instruction throughput is higher, $\frac{2 \mathrm{insn}}{(5-2) \mathrm{cyc}}=\frac{2}{3} \mathrm{insn} /$ cycle based on the second iteration starting at cycle 2 and the third iteration starting at cycle 5 .

```
LOOP: # Code in Static Instruction Order
    bne r1, r2, LOOP
    addi r1, r1, 4
    xor r5, r6, r7
    sub r8, r9, r10
    # SOLUTION -- Dynamic Instruction Order
LOOP: # Cycle 0
    bne r1, r2, LOOP IF ID EX ME WB # First Iteration
    addi r1, r1, 4 IF ID EX ME WB
LOOP: # Cycle 0
    bne r1, r2, LOOP IF ID -> EX ME WB # Second Iteration
    addi r1, r1, 4 IF -> ID EX ME WB
LOOP: # Cycle }0
    bne r1, r2, LOOP IF ID -> EX ME WB
    addi r1, r1, 4 IF -> ID EX ME WB
```



Show execution until the fetch of the lw r1 in the second iteration.
 Show instruction throughput (IPC) assuming a large number of iterations.

Solution appears below. The add stalls due to the dependence carried by $r 3$ and the sw stalls due to the dependence carried by $r 4$. In this implementation there is a bypass to the memory port D In connection and so the sw r4 need only stall one cycle. In cycle 1 the sub and sw r5 are stalling only so that instructions in ID remain in program order.

The instruction throughput is $\frac{9 \mathrm{insn}}{(7-0) \mathrm{cyc}}=\frac{9}{7} \mathrm{insn} /$ cycle.

```
    # SOLUTION -- Dynamic Instruction Order
LOOP: 0
    lw r1, O(r2) IF ID EX ME WB # 1st Iteration
    lw r3, 4(r2) IF ID EX ME WB
    add r4, r1, r3 IF ID ----> EX ME WB
    sw r4, O(r6) IF ID -------> EX ME WB
    sub r5, r1, r3 IF -------> ID EX ME WB
    Sw r5, 4(r6) IF -------> ID -> EX ME WB
    addi r6, r6, 4 IF -------> ID -> EX ME WB
    bne r2, r9, LOOP IF -------> ID -> EX ME WB
    addi r2, r2, 8 IF -> ID EX ME WB
    xor r10,r11,r12
    IFx
LOOP: 0
    lw r1, O(r2) IF ID EX ME WB # 2nd Iteration
```

Problem 2: (20 pts) Appearing below is our MIPS implementation with a floating-point pipeline. The select inputs of some multiplexors are labeled with a letter. Also, the inputs to some multiplexors have been colored to make them easier to follow.

$\nabla$ Show the values on the select inputs (D, E, and H) expected from the execution shown below. $\nabla$ Leave a signal blank if it does not affect execution.

| \# Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| \#\# SOLUTION |  |  |  |  |  |  |  |
| D: |  |  | 2 | 0 | 3 |  |  |
| E: |  |  | 1 | 2 | 2 |  |  |
| H: | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| \# Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| \# Cycle | 0 |  |  |  |  |  |  |
| add R3, ry, r6 | IF | ID | EX | ME | WB |  |  |
| add r2, R3, 4 |  | IF | ID | EX | ME | WB |  |
| lw ri, 0(R3) |  |  | IF | ID | EX | ME | WB |
| \# Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

$\nabla$ Show instructions that could have produced the select input (D,E,G, and H) values shown below. $\nabla$ Take dependencies into account when choosing register numbers.

| \# Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| D: |  |  | 2 | 2 | 3 |  |  |
| E: |  |  | 2 | 1 | 2 |  |  |
| G: | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| H: Cycle |  |  |  |  |  |  |  |
| \# Cycle |  |  |  |  |  |  |  |
| \#\# SOLUTION | IF | ID | EX | ME | WB |  |  |
| lw R3, 1(r5) |  | IF | ID | EX | ME | WB |  |
| add R2, rf, rt |  |  | IF | ID | EX | ME | WB |
| SW RT, O(R3) | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| \# Cycle |  |  |  |  |  |  |  |

Show the values on the select inputs expected from the execution shown below. $\square$ Leave an input blank if it does not affect execution.

| \# Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| \#\# SOLUTION |  |  |  |  |  |  |  |  |  |  |
| G: |  |  |  | 2 |  |  |  |  |  |  |
| H: |  |  |  |  | 0 |  | 1 |  |  |  |
| C: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| \# Cycle |  |  |  |  |  |  |  |  |  |  |
| lwei fl, 0(r5) | IF | ID | EX | ME | NF |  |  |  |  |  |
| Swc1 fa, 0(r7) |  | IF | ID | EX | ME | WB |  |  |  |  |
| mic fa, rB |  |  | IF | ID | EX | ME | NF |  |  |  |
| adds fy, ff, f6 |  |  | IF | ID | Ai | A2 | AS | A4 | VF |  |
| \# Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Problem 3: (20 pts) Appearing to the right is the early writeback 2-way superscalar implementation from the 2021 Final Exam and 2022 Homework 6. Recall that in this implementation if slot 1 contains a load instruction then slot 1 writes back when it reaches WB but slot 0 writes back early, in ME/MW. If slot 1 does not contain a load instruction slot 0 writes back when it reaches WB and slot 1 writes back in ME/MW. This is illustrated in the execution below.

```
# Cycle 0
add r2, r3, r4 IF ID EX ME WB # Slot 0 uses WB since slot 1 isn't a load.
and r1, r5, r6 IF ID EX MW
or r10, r9, r7 IF ID EX MW # Slot 0 uses MW since slot 1 is a load.
lw r6, 8(r11) IF ID EX ME WB
# Cycle 0
```

(a) Consider the execution of the code below:

```
# Cycle 0
add r2, r3, r4 IF ID EX ME WB
sub r1, r2, r5 IF ID -> EX MW
```

The sub stalls because it needs to wait for r2 from the add. Add control logic to generate a stall when slot 1 depends on slot 0 . The output of rs src and rt src are 1 when the slot- 1 instruction uses the rs and rt registers as sources. Use these in your solution.
$\checkmark$ Add hardware to generate a stall (see the big OR gate) when slot 1 depends on slot 0 .
The solution is on the next page.
(b) In the first code fragment below the lw r5 stalls, but because the lw has a zero immediate it could have just used the value computed by the add instruction (since there is no need to add anything to it). In the second execution Mux A is used to perform a lateral bypass from the add to the 1 w during cycle 2, avoiding the stall.

Modify the control logic so that a lateral bypass will be performed when there is a zero-immediate load in slot 1 that depends on the slot- 0 instruction. Other code, such as the examples at the top of this problem, should continue to work correctly.

```
# Cycle 0
add r2, r3, r4 IF ID EX ME WB
lw r5, O(r2) IF ID -> EX ME WB
# Cycle 0
add r2, r3, r4 IF ID EX MW # add executes normally.
lw r5, O(r2) IF ID EX ME WB # Lateral Bypass: lw uses slot-0 alu value.
```

$\checkmark$ Add logic to detect whether a lateral bypass is possible, and if so, suppress the stall from part a.
$\checkmark$ Modify the control logic to implement a lateral bypass, in part using Mux A.
$\boxed{\nabla}$ Make sure that $\nabla$ early writeback continues to work correctly in other cases and $\nabla$ that the destination of the slot 0 and slot 1 instructions are written to the correct registers.
$\boxed{\text { As engineers always do, pay attention to cost and performance. }}$
The solution is on the next page.

Solutions appears below. The hardware for part (a) appears in purple. To detect the dependence the destination of the slot-0 instruction is compared to the $r s$ and $r t$ sources of the slot- 1 instruction. The rs sre and rt sre blocks are used to check whether the rs and $r t$ fields of the instruction are used for sources. (In most type I instructions the rt is not used for a source. Some instructions, such as floating-point instructions don't have any integer sources.)

The hardware for part (b) appears in orange. A lateral bypass is possible if there is a memory instruction in slot 1 with a zero immediate, and if the rs source depends on the slot-0 instruction. If these conditions are true the stall is suppressed.

The logic for implementing the lateral bypass is very simple. If there is a lateral bypass Mux A uses input 0 , otherwise Mux A uses the input it would have used without a lateral bypass.


Problem 4: (20 pts) Answer the following branch prediction questions.
(a) Code producing the branch patterns shown below is to run on several systems, each with a different branch predictor. One system has a bimodal predictor and the other system has a local predictor with an 6 -outcome local history.
Answer each question below, the answers should be for predictors that have already warmed up. Show work or provide brief explanations.


What is the accuracy of the bimodal predictor on branch B1?Be sure to base the accuracy on a repeating pattern.

What is the accuracy of the local predictor on B1 ignoring B2.

What is the accuracy of the local predictor on B2 ignoring B1.

What is the accuracy of the bimodal predictor on branch B2?

What is the shortest history size for which the local history predictor is better than the bimodal predictor on branch B2?

Problem 5: ( 20 pts ) Answer each question below.
(a) The diagram below shows a simple direct-mapped cache and the address bit categorization of four lookup addresses ( $0 \times 5439,0 \times 1270, .$.$) .$


Two kinds of memory are used in the diagram above, fast/expensive and slow/cheap.
$\nabla$
On the diagram above show which blocks are fast and which blocks are slow.
The blocks are labeled in blue. The Memory block is labeled slow. (If it weren't slow there would be no need for a cache.) Note that both the Data Store and Tag Store must use fast memory.

Suppose that the cache is initially cold (there is nothing in the cache). Show the outcome, hit or miss, of each of the four lookups.

Show outcome, hit or miss, on diagram above.
The outcomes appears in the diagram above, in blue.

Find the addresses requested below.
After the four lookups, what is the smallest address that will hit the cache.
The smallest address is $0 \times 1270$.
$\checkmark$ After these four lookups, what is the largest address that will hit the cache.

## Answer: The largest address is $0 x 643 f$.

Explanation: The largest lookup address is $0 \times 6439$. Since it is the last of the four lookup addresses it will surely be in the cache after the four lookups are complete. Each cache miss, including the one for lookup address 0x6439, brings in a line's worth of data. The starting address of a line is found by setting the offset and align bits (bits 3 to 0 here) to zero. For $0 x 6439$ the line starting address is $0 \times 6430$. The last, or largest, address in a line can be found by setting all of the offset and align bits to 1 . That yields the answer to the question, $0 x 643$.
(b) Show the encoding for the beq and lw as used in the code below. Be sure to include the immediate value.

```
addi r6, r0, 10
beq r2, r6, SKIP
lw r1, 4(r3)
add r1, r1, r5
SKIP:
and r9, r9, r1
```

$\square$ Encoding of beq. $\boxed{\nabla}$ Be sure to show a value for the immediate field.
The encoding is shown below. Though the solution shows the opcode of beq, $100_{2}=4$, full credit would be received for an answer that showed beq or something like that in the opcode field.

$\checkmark$ Encoding of lw. $\nabla$ Be sure to show a value for the immediate field.
The encoding is shown below. Though the solution shows the opcode of $1 \mathrm{w}, 10011_{2}=23_{16}=35_{10}$, full credit would be received for an answer that showed lw or something like that in the opcode field.

(c) Answer the following about ISA families.
$\boxed{\checkmark}$ Which style of implementation are RISC ISAs designed for?
RISC ISAs are designed for pipelined implementations.
$\checkmark$ Which style of implementation are VLIW ISAs designed for?
VLIW ISAs are designed for multiple issue (sort of like superscalar) implementations.
(d) Some early RISC ISAs omitted useful magnitude-comparison branch instructions such as bgt r1, r2, TARG in which the branch is taken if r1 > r2. As branch prediction became more common magnitudecomparison branch instructions were added to RISC ISAs. One might argue that with branch prediction the cost and performance impact of magnitude-comparison instructions was lower.
$\checkmark$ Explain how the cost of implementing bgt is lower with branch prediction than without.
Because without branch prediction it is likely that the branch would be resolved in the ID stage and so would require the use of a magnitude comparison unit used only for resolving branches. With branch prediction the magnitude comparison could be done later, in EX, and could be done using the ALU, and so no extra magnitude comparison unit would be needed.

Explain how the performance impact of implementing bgt is lower with branch prediction than without.
Without branch prediction the magnitude comparison would likely be done in the ID stage. That magnitude comparison could not start until the register values were retrieved from the register file, and the time to do both might lengthen the critical path, and so lower performance. With branch prediction the register values would be retrieved in one cycle ID, and the comparison would be done in the next cycle, EX, assuming something like the MIPS five-stage implementation used in class.

